PEARSON

HZ BOOKS

The Process of Software Architecting

架构实战
软件架构设计的过程

（英）Peter Eeles　Peter Cripps 著
蔡黄辉 马文涛 译

机械工业出版社
China Machine Press

PEARSON Education

The Process of Software Architecting

架构实战

软件架构设计的过程

成功的软件离不开好的软件架构，高效的架构设计需要透彻地理解组织的角色、工件、执行的活动以及执行这些活动的最优顺序。

本书介绍了如何应对软件系统架构设计时的各种挑战，引入了基于Java EE、Microsoft .NET或其他技术的最佳实践。书中首先阐述了架构设计文档、可重用资源等软件架构的相关概念，接着通过一个典型项目介绍了一个容易理解的、关注任务的旅游指导（这个项目关注架构师的角色），并讨论了一些常见问题，最后总结了一组可以应用于当今最复杂系统的最佳实践。

本书适合软件架构师、项目经理和软件从业人员阅读。

本书主要内容：

- 在典型的软件开发项目中架构师扮演的角色
- 如何编写软件架构文档来满足不同利益相关者的需求
- 架构设计过程中可重用资源的适用性
- 在定义需求时架构师扮演的角色
- 如何基于一组需求来获取架构
- 创建复杂系统的过程中架构设计的相关性

作者简介

Peter Eeles　IBM Rational Software的高级IT架构师，主要工作是进行架构设计和实现大规模、分布式的系统。他目前致力于帮助组织提高软件开发能力。除本书外，Eeles还与人合作编写了《Building J2EE™ Applications with the Rational Unified Process》（Addison-Wesley，2003）和《Building Business Objects》（Wiley，1998）。

Peter Cripps　IBM Global Business Services的高级IT架构师，专注于应用组件和基于服务的开发技术，并在整个IBM公司推广架构设计最佳实践，目前从事IBM Unified Method Framework的开发工作。

PEARSON
www.pearsonhighered.com

客服热线：(010) 88378991, 88361066
购书热线：(010) 68326294, 88379649, 68995259
投稿热线：(010) 88379604
读者信箱：hzjsj@hzbook.com

华章网站 http://www.hzbook.com

网上购书：www.china-pub.com

封面设计：王建敏

上架指导：计算机/软件工程

ISBN 978-7-111-30115-8

9 787111 301158

定价：45.00 元

The Process of Software Architecting

架构实战
软件架构设计的过程

（英）Peter Eeles　Peter Cripps 著
蔡黄辉 马文涛 译

机械工业出版社
China Machine Press

本书从基本原理入手，介绍软件架构设计过程中涉及的一些概念、流程、方法、用到的工作产品及可重用的资源，从第6章开始，通过介绍一个具体的案例来阐述如何定义需求、创建逻辑架构、创建物理架构。在第10章“进阶”中，作者补充说明了架构师和软件开发项目其他方面的关系，后面又说明了各种软件开发项目可能存在的困难及相应的处理方法。

本书理论结合实践，介绍了一些可以应用到整个或部分的架构设计流程中的最佳方法。不管你是一位资深的架构师还是一位有志于成为架构师的初级使用者，通过阅读本书都能从中获益。

本书版权登记号：图字：01-2009-5640

图书在版编目(CIP)数据

架构实战：软件架构设计的过程/(英)伊乐斯(Eeles,P.),克里普斯(Cripps,P.)著;蔡黄辉,马文涛译. —北京：机械工业出版社,2010.4

书名原文：The Process of Software Architecting

ISBN 978-7-111-30115-8

Ⅰ.架…　Ⅱ.①伊…　②克…　③蔡…　④马…　Ⅲ.软件设计　Ⅳ.TP311.5

中国版本图书馆 CIP 数据核字（2010）第044629号

机械工业出版社（北京市西城区百万庄大街22号　邮政编码　100037）

责任编辑：张少波

北京市荣盛彩色印刷有限公司印刷

2010年4月第1版第1次印刷

186mm×240mm · 16印张

标准书号：ISBN 978-7-111-30115-8

定价：45.00元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88378991；88361066

购书热线：（010）68326294；88379649；68995259

投稿热线：（010）88379604

读者信箱：hzjsj@hzbook.com

对本书的赞誉

“软件架构师这个角色在最近几年很盛行，也被认为是项目成功的一个关键因素。然而，即使在今天，人们对如何分析需求、理解关注点、评估可选方案及构建和编写符合目的的架构描述文档等工作仍然缺少一些常规的理解。Eeles 和 Cripps 在他们这本非常有用和有实践性的书中填补了这个空白。书中的内容清楚易懂，遵循从起始到交付的一个逻辑流程，通过研究一个真实的案例对任务和工作产品进行了清楚的解释和阐述。无论对于新的架构师，还是经验丰富的专家，这都是一本重要的手册。”

——Nick Rozanski，《软件系统架构》的作者之一

“如果您需要一本关于软件架构流程的全面和权威的参考书，那就不用再等待了。Peter Eeles 和 Peter Cripps 已经为这个流程编写了一本权威性的指导参考书。本书中介绍的流程利用一个元模型进行了准确的定义，通过一个真实的研究案例进行了阐述，还清楚地关联到像 UML、RUP 和 IEEE 1471 等这样的关键标准，因此为那些大型项目开发中的软件架构提供了颇有价值的指导。我一点都不怀疑本书会成为许多软件架构师的一本很有价值的参考书。”

——Eoin Woods，《软件系统架构》的作者之一

“Eeles 和 Cripps 把多年的经验汇集到这本指导书中，帮助读者不仅理解架构师生产什么，还理解他们如何生产。本书是一本具有很高实践性的指导书，其中详尽阐述了获得的经验和需要避免的陷阱。已经成为架构师的人将参考本书，因为它能够使他们的技术更完善；而期望成为架构师的人通过阅读它能够获得一些需要多年痛苦的经历才能获得的关键见识。”

——Bob Kitzberger，IBM Software Group 的程序主管、战略家

“就我在这个领域的工作经验来看，软件架构给人的感觉有点像妖术，只有精选的少许专家和天才才有天分从事这项工作。本书先介绍行业最佳实践和作者宝贵的经验，然后把架构解决方案带入一个真实的工程学科的范畴。现在，我有了一本可以传授给新从业者的参考书，一本讲授过去需要多年尝试和出错才能体会到的经验的书。”

——Colin Renouf，英国 Websphere User Group 的副主席，企业架构师和技术作家

译 者 序

光阴荏苒，时光如逝，一转眼，我已经工作十年有余。在这十年间，我基本上一直在从事软件开发的工作。从编码到设计，再到需求，再参与一些管理工作。我相信大部分从事软件行业的人都和我有相似的经历。在这个过程中，大家都会时不时地思考一些和软件开发相关的问题：什么是软件项目？软件项目的目的是什么？如何在资源（人力、成本、时间等）有限的情况下尽可能地开发出高质量的软件产品？架构在软件项目的整个过程中起到什么作用？架构师在软件项目中担任什么角色，能起到什么作用？等等。对于这些问题，在本书中都能找到答案。

本书从基本原理入手，介绍软件架构设计过程中涉及的一些概念、流程、方法、用到的工作产品及可重用的资源，从第 6 章开始，通过介绍一个具体的案例来阐述如何定义需求、创建逻辑架构、创建物理架构。在第 10 章“进阶”中，作者补充说明了架构师和软件开发项目其他方面的关系，后面又说明了各种软件架构设计过程中可能存在的困难及相应的处理方法。总的来说，本书理论结合实践，介绍了一些可以应用到整个或部分的架构设计过程中的最佳方法。不管你是一位资深的架构师还是一位有志于成为架构师的初级使用者，通过阅读本书都能从中获益。

本书的第 1 ~5 章由我翻译，第 6 ~10 章由我的朋友马文涛翻译。由于时间仓促，书中有些处理不妥的地方，敬请谅解。

最后，我要感谢我的父母和妻子，感谢他们在我翻译的过程中所给予的帮助和支持。

蔡黄辉

2010 年 3 月 5 日

序

软件密集型系统的架构研究目前受到很多关注。虽然已经出现了大量描述软件架构概念的书籍和论文，但是其中很少是描述架构实践的。作者介绍的就是这样一本书。

在本书中，作者提出了他们关于架构的适用的观点：什么是架构？它是如何证明自己的？谁设计的？它是如何与一个软件开发项目的其他工件关联起来的？尽管本书包含了描绘一个系统架构的所有重要的基本原理，但是，本书的重点在于架构方法的深入验证。我说“深入”是因为作者通过使用定义良好的模型来解释他们的方法，除此之外，令人高兴的是，他们采用的方式很容易理解，而且很全面，特别注意组成一个系统架构完整构想的各种观点。具体地说，他们讲到了在一个系统的逻辑架构和物理架构之间建起桥梁。令我尤其高兴的是，他们还包含了人件问题——也就是，架构师和架构团队的角色，以及各个团队成员活动的架构工件的联系。

通过一个相当完整的案例研究，作者将他们的观念运用到了实践中。来自于他们在行业中的实践经验的、人们应该避免的缺陷的解释特别有用。毫无疑问，这是一本非常有用的书，因为它包含了任何软件开发组织可以立即运用的观点。

Grady Booch

前　言

几年前，作者们开始注意到 Grady Booch 首创的《软件架构手册》（《Handbook of Software Architecture》，www. handbookofsoftwarearchitecture. com）。Grady 起初的目的是：

> 整理许多有趣的软件密集型系统的架构，以揭示它们的基本模式以及允许在域和架构风格之间进行比较的方式，并把它们呈现出来。

当 Grady 正关注于最终架构的时候，我们感到理解成功架构师创建他们的架构时所遵循的流程同样很有趣。当然，我们最终的目的是复制他们的成功。我们花了好几年的时间才完成这个过程。我们做了许多项目，和许多架构师进行了交流，还对许多开发方法进行了梳理——所有这些都有助于我们理解当构建一个软件系统时起作用和不起作用的因素的本质。本书是我们经历的这个过程的总结。

许多优秀的书讲述了软件架构过程的特定方面，我们借鉴了这些书中的相关内容。例如，其中有些书注重编写一个软件架构的文档，另一些书注重如何评价一个软件架构。其中任意一方面都适合比较大的场景，因为每一方面都呈现了软件架构过程中的一个重要因素。因此，本书的一个目的是通过提供在一个典型软件开发项目的环境中架构的所有方面的概览来呈现这个大场景。

应该指出的是，本书没有指定一个专门的软件开发方法。更确切地说，本书讲述了人们在支持构建过程的任意现代开发方法中可能遇到的关键因素。

本书是为谁准备的

显然，本书是针对那些想了解他们的角色如何适应整个软件开发过程的软件架构师（或者立志成为软件架构师的人）。本书也适合“专门”的架构师角色，如应用架构师和安全架构师。更笼统地说，本书适合那些想更好地了解软件架构师这个角色的人。就这点而言，它对一个软件开发团队的所有成员也都有好处，包括开发人员、测试人员、业务分析人员、项目经理、配置经理和过程控制工程师。本书也特别适合那些在软件开发的尝试中想了解日益重要的软件架构师角色的大学生。

如何阅读本书

本书大致分为三个部分：

第 1 ~5 章为第一部分，概述了架构、架构师、架构设计、编写软件架构文档、可重用架构资源的核心概念。

第 6 ~9 章为第二部分，这部分包含了相关案例研究的章，通过一个基于样例应用程序的典型软件开发项目，重点体现架构师这个角色，提供一个指导性指南。这些章的编写方式，使你很容易找到感兴趣的特定主题。每个相关案例研究的章主要按照任务进行组织，另外，在这些章中我们使用了一些排版体例。特别是，对流程元素的所有引用，如任务、工件和角色，都用黑体加以强调，例如当我们描述**软件架构文档**工件的时候。

第 10 章为第三部分，包含额外讨论的话题和思考，尤其是，在前面的章中描述的概念如何应用于架构复杂的系统。

在本书中，你还会发现一些如下进行分类的有用的补充内容：

- 概念补充内容：介绍与讨论与主题相关的想法或整套想法。
- 检查清单补充内容：包含当执行某一特定任务时，可以进行检查的有用的项目。
- 最佳实践补充内容：介绍已经在实践中证实为有效的方法。
- 缺陷补充内容：介绍最好避免的方法，因为它们会导致负面效果。

我们在本书中大范围地使用统一建模语言（UML）来描述架构的某些方面。所有的 UML 图表都是通过 IBM Rational Software Architect 创建的。

附属站点

本书有一个附属的站点：processofsoftwarearchitecting. com，读者可以在这个站点上找到额外的信息，也可以和作者进行交流。

致　　谢

当编写本书的时候有两位关键人物参与进来了。一位是 Grady Booch，他不但为本书奠定了基础，而且还友好地为本书写了序，提供了详细的审稿意见及自始至终的支持。另一位是 Philippe Kruchten，他从未停止过令人惊奇的经历。此外，Philippe 通过提供许多建议，担当被访问者和评论家，在编写本书的过程中为我们引见了很多架构师进行交流等，对本书提供了自始至终的支持。

同样地，我们还要感谢 Brad Appleton、Dave Braines、Alan Brown、Mark Dickson、Celso Conzalez、Holger Heuss、Bobby Higgins、Rich Hilliard、Kelli Houston、Brad Jackson、Robert Kitzberger、Colin Renouf、Nick Rozanski、Rick Smith 和 Eoin Woods 为本书提供了详细的审稿意见。有了他们，本书才变得更加完善。

在访谈的所有架构师和项目经理中，有人放弃大量的个人时间与我们一起工作。他们是 Ian Charters、Philippe Kruchten、Helen McCann、Gordon McClean 和 Nick Rozanski。

我们还要感谢众多的客户以及那些支持我们的工作从而也支持了本书的同事和熟人。很遗憾，有太多的人要感谢以致不能一一提及，但是我们要特别感谢我们许多的 IBM 同事、电机及电子学工程师联合会/IFIP 软件体系结构（WICSA，Working IEEE/IFIP Conference on Software Architecture）会议的参与者、国际信息处理联合会（IFIP，the International Federation for Information Processing）2.10 工作组（软件架构）的成员以及英国计算机协会（BCS，British Computer Society）软件实践改进（SPA，Software Practice Advancement）专家组。这些会议和讨论会专门提供机会让我们在行业和学术界中的重要人物面前得以展现。

最后，我们还要感谢培生教育（Pearson Education）中参与这个项目的所有人。我们特别要感谢我们的组稿编辑 Chris Guzikowski，还有 Raina Chrobak、Sheri Cain 和产品团队。

作 者 简 介

Peter Eeles 是 IBM 的高级 IT 架构师，他就职于 IBM 的 Rational 品牌软件组。在这个职位上，他帮助组织提高软件开发能力，尤其关注和致力于改进架构流程。Peter 从 1985 年开始从事软件行业，其主要工作是进行架构设计和实现大规模、分布式的系统。Peter 是《Building J2EE Applications with the Rational Unified Process》（Addison-Wesley，2002）和《Building business Objects》（John Wiley & Sons，1998）的合著者。他还是英国计算机协会高级会员（FBCS）、工程技术协会（FIET）会员、IBM 技术人员、Open Group Master 认证的 IT 架构师、特许 IT 专家（CITP）。

Peter Cripps 是英国 IBM Global Business Services 的一位 IT 架构师。他从 1980 年开始从事于软件行业，在这期间，他当过程序员、实时软件工程师和跨多个行业（包括电信、财经服务、零售和政府部门）的流程工程师。Peter 的技术特长领域及兴趣是基于组件和服务的开发技术的运用以及优秀架构方法的开发。他是英国计算机协会会员（MBCS）和特许 IT 专家（CITP）。

目　录

第 1 章

导　　言

Bjarne Stroustrup 是 C ++ 程序设计语言的发明者，他曾经说：“我们的文明建立在软件之上。”软件实际上已经渗入到我们日常生活的许多方面，从简单得像用普通手机打开的唱“生日快乐”的生日卡片，到像飞机和核电站这样非常复杂的系统，都可以看到软件。实际上，如果没有软件，今天我们认为是理所当然的许多创新和像 eBay 或者 Amazon 这样的组织都不会存在。即使是传统的组织，如那些在金融、零售和公共部门中的组织，也很大程度地依赖软件。在如今这个时代，很难找到一个不以某种方式使用软件处理业务的组织。

架构设计流程就是在这种情况下开始形成的。如果希望目前这种对软件日益依赖的情形得以延续，那么，软件必须具有必需的功能、足够好的质量、在承诺的时间内可用，还要以可接受的价格交付。所有这些特性都直接受软件架构的影响，由此得出结论，假如我们把软件架构设计好，那我们就离期望的目标不远了。

本书旨在指导您体验那些在设计一个软件系统的架构的过程中采用的任务和最优方法。在达到这个目标的过程中，本书将回答一些基础性的问题，如下所示：

- 什么是架构？
- 在一个软件开发项目中架构师的角色是什么？
- 架构师与项目的其他角色是什么关系？
- 关于需求，架构师的角色是什么？
- 如何描述架构？
- 架构师在什么时候以及如何才能生成一个初始的架构？
- 架构师如何精炼架构？
- 架构师什么时候会考虑重用？
- 架构是如何验证的？

1.1　流程应用

在回答上述问题的过程中，我们通篇介绍了一系列在角色、任务和工作产品中经过证实的最佳方法。

> **实践**（practice）是解决一个或者多个经常发生的问题的方法。这些方法也有意作为采用、启动和配置的流程“块”（chunk）。（RUP 2008）

这些方法可以合并到您可能使用的任何流程中，包括瀑布式流程、迭代开发流程（如统一软件开发过程），或者敏捷流程（如 Scrum）。同样地，我们没有把这些方法和任意特定的流程联系起来，虽然我们确实按这样的一个顺序来介绍这些方法：该顺序允许我们在项目生命周期的不同时刻展示它们之间的关系和相关性。我们也鼓励您挑选那些能给您的项目（当然还有您的架构）带来最大价值的方法。

1.2 流程概述

为了达到这个目标，我们认为在本书中提供一个描述关键流程元素的概览图来提高您的兴趣是值得的。在这里，我们仅关注本书中详细讲述的任务的单次流程，虽然在一个项目中会把每一个任务执行好多次（我们把这种情况称为迭代）。

组成这个流程的宏观活动的高级概览如图 1.1 所示，这个图与软件工程方法相一致。

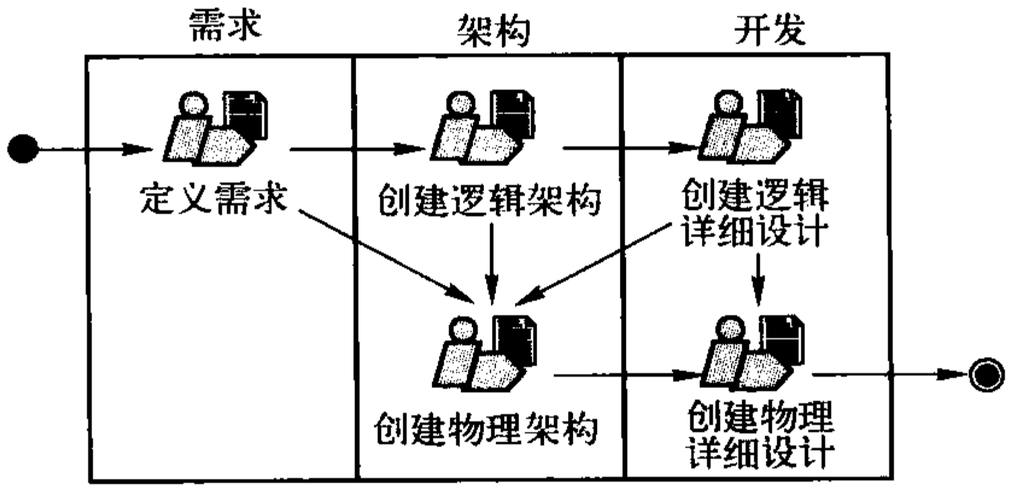


图 1.1 活动概览

> 这是把定义主要“利益范围和/或协作效率”的任务组织起来的最初分类机制的一个规范。（OpenUP 2008）

正如您所看到的，架构设计活动位于需求和开发的中间。迭代从**定义需求**这个活动中的软件需求定义开始。虽然这个活动主要是业务分析人员的职责，但是，架构师也会参与这个活动的一些详细任务。随后，架构师在**创建逻辑架构**这个活动中创建一个大概的架构。这时候产生的架构不考虑任何技术因素，称为逻辑架构。我们可以把这一步当作是从需求到物理架构的一个跳板，物理架构是针对特定技术的，它组成了实现（编码）的基础。这个逻辑架构会作为**创建逻辑详细设计**活动中执行的任何详细设计的输入。创建逻辑详细设计这个活动会生成在逻辑层面上必需但对架构不必需的任何余留细节的扩充。关于架构与详细设计之间的区别的说明，请参考后面的“概念：架构与设计的比较”部分。

在需求、逻辑架构和逻辑详细设计的基础上，架构师接着在**创建物理架构**这个活动中精炼这种架构，创建物理架构这个活动需要考虑技术因素，最终产生物理架构。这个物理架构会作为**创建物理详细设计**活动中执行的任何详细设计的输入，创建物理详细设计这个活动会形成实现的基础。虽然详细设计和实现这两个活动不是架构师的职责，但是，在需要的时候他们必须给这些团队提供指导。

概念：架构与设计的比较

所有的架构都是设计，但并不是所有的设计都是架构。架构代表塑造一个系统的重要设计决策，这里的重要性通过改变所需要的成本来衡量。（Booch 2009）

换句话说，架构可以看作是战略上的设计，而详细设计可以看作是战术上的设计。

连接活动的箭头不是想表示一个必须遵守的顺序——仅仅是一个比较合适的顺序。例如，基于执行的架构设计，您可能发现必须对需求进行说明，从而最终可能重新回到需求。

现在让我们略微深入地看一下其中的部分活动，以便您可以更好地理解在这个迭代过程中的架构师角色。图 1.2 中显示了组成**定义需求**这个活动的详细任务。这个迭代从**收集利益相关者的需求**这个任务开始，正如这个任务的名称所示，它关注于了解各种各样的利益相关者的需求。这些需求为架构师提供了需要设计系统的范围的一个初始的指示。在**整理常用词汇**这个任务中的术语表完成后，架构师特别感兴趣的就是**定义系统上下文**这个任务，因为它定义了必须与系统交互的外部元素，如最终用户和其他系统。在这个上下文的基础上，**概要说明功能性需求**和**概要说明非功能性需求**这两个任务分别说明功能性需求和非功能性需求。架构师不仅对关键的功能性需求感兴趣，还对系统质量（如性能）和解决方案约束（包括非功能性需求）感兴趣，处理这些非功能性需求通常比处理功能性需求更有挑战性。

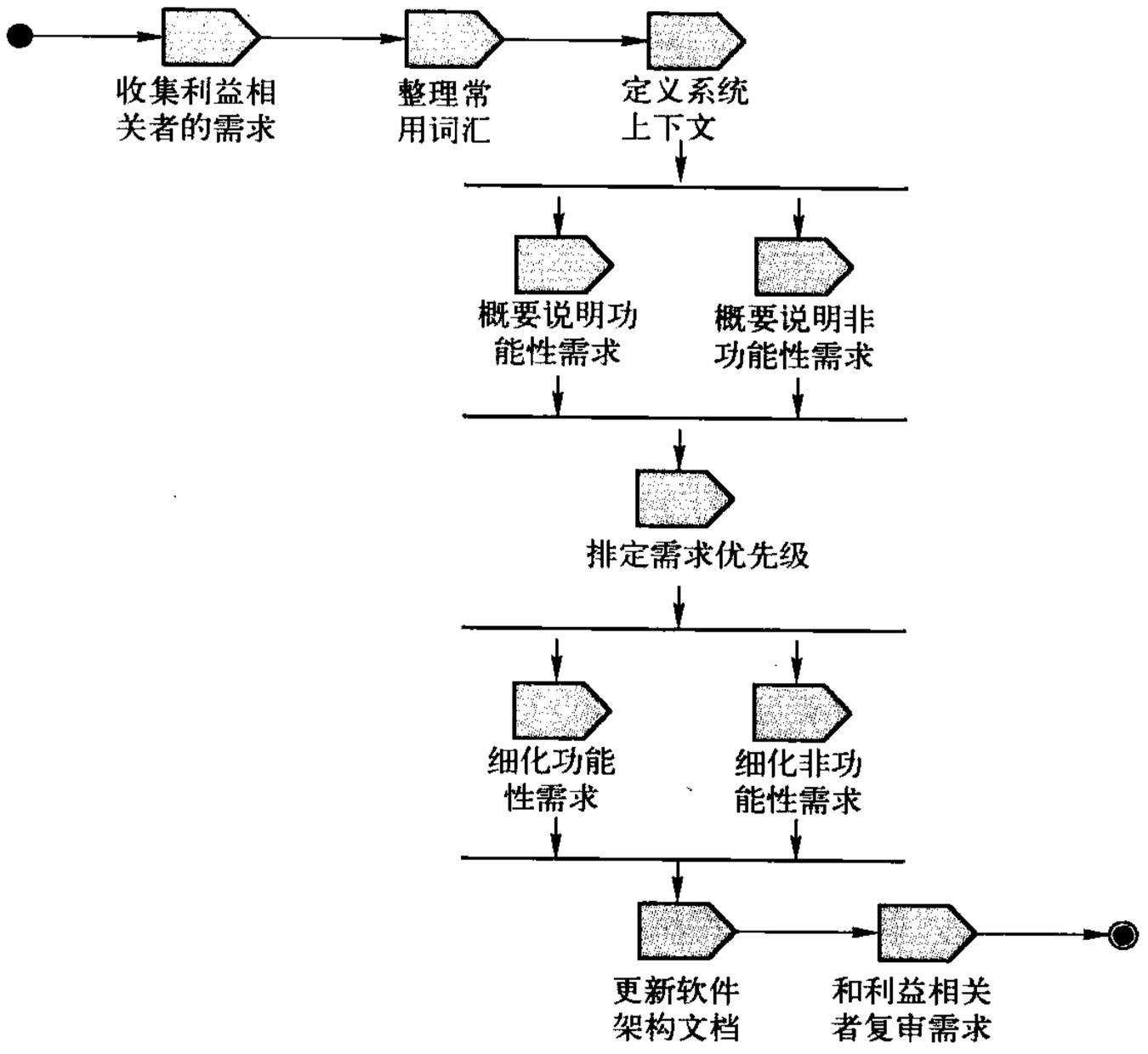


图 1.2　定义需求活动中的任务

从某种程度上，架构师参与整个**定义需求**活动以确保需求能够按通过可用的技术、在指定的时间和预算内就可以实现的方式来指定。这通常要求与利益相关者进行一定程度的沟通，架

构师也要积极地参与进来。就我们关心的需求任务而言，**排定需求优先级**这个任务与架构师关系特别大，在这个任务中，架构师要保证优先级受那些能使架构尽可能快速稳定的需求和风险的影响。

排定了优先级的需求推动了其余内容的迭代，在这种意义上，只有最高优先级的需求会被考虑，因为您处理的需求范围将决定您是否会有一个可行的系统。（低优先级的需求会在随后的迭代中进行考虑。）高优先级的需求会在**细化功能性需求**和**细化非功能性需求**这两个任务中进行细化。接下来，架构师会在**更新软件架构文档**任务中正式地编写架构上重要需求的概要文档。这些与需求相关的任务最终以**和利益相关者复审需求**这个任务结束。就活动来说，连接活动的箭头不是表示一个必须遵守的顺序（仅仅是一个比较合适的顺序），在需要时可以重新回到这些任务。

组成**创建逻辑架构**这个活动的任务如图 1.3 所示，在**调查架构资源**这个任务中，架构师始终考虑如何使用现有的资源。即使在项目的早期，作为一个架构师，您都可能会选择一个对您的架构有明显影响的资源，如一个参考架构。当执行每个任务时，架构师都在**编写架构决策的文档**这个任务中记录那些决策。

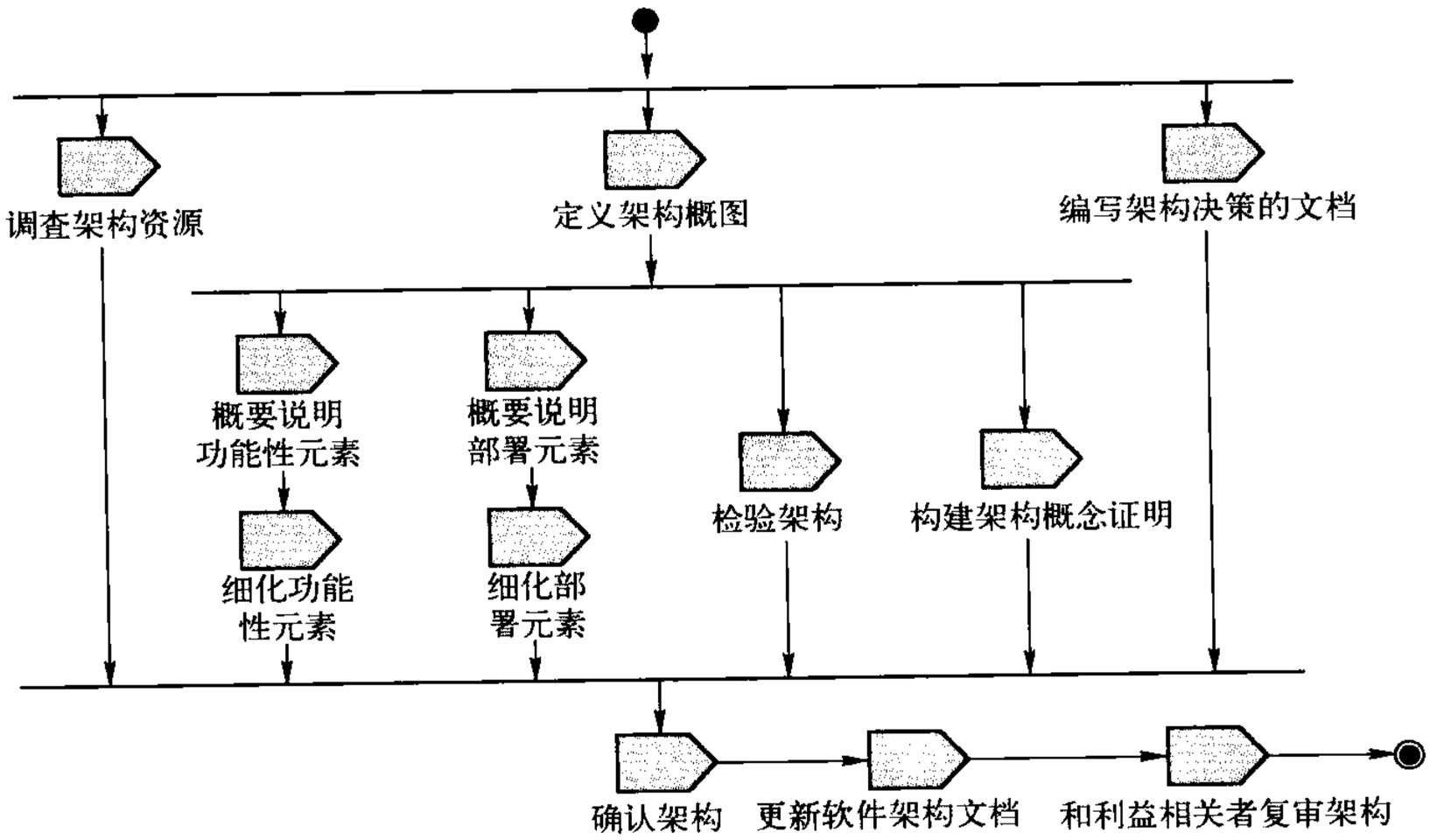


图 1.3 创建逻辑架构活动中的任务

基于最高优先级的需求，架构师在**定义架构概图**这个任务中概要说明候选的解决方案。架构概览提供了架构的“整体”轮廓。然后，**概要说明功能性元素**和**概要说明部署元素**这两个任务同时执行，接着根据组件、它们的交互和关系以及它们在节点上的部署精炼这个轮廓。**检验架构**这个任务确保组成架构的各种功能性元素和部署元素相互一致，尤其是确保跨这些元素的任何要点（如既影响功能元素又影响部署元素的性能特性）都被适当地处理。我们承认功能性元素和部署元素未必支配每个架构（例如，并行元素可能会支配一个实时、内嵌的系

统），因此，我们在第 4 章中介绍了一个可展开的架构描述框架以适应这样的情况。

在**构建架构概念证明**这个任务中架构师还关注证明架构的某些方面。这样做的目的是合成至少一个满足架构上重要需求的解决方案（可能仅仅是概念上的解决方案，而不是逻辑上的解决方案）来确定像架构师设想的这样一个解决方案是否存在。特别地，概念证明提供了一个工具来减少某些与架构相关的风险，而且经常表现为允许系统功能和特性得到评估的可执行软件的形式。

接下来，架构元素在**细化功能性元素**和**细化部署元素**这两个任务中进行细化。**确认架构**这个任务确保架构元素会像声明的那样满足需求（包括功能上的需求和非功能上的需求），还确保项目上的一些考虑，如资源、预算和进度约束。然后，架构师在**更新软件架构文档**这个任务中编写一个不依赖平台的架构的概要。由此形成的架构描述用来和相关的利益相关者沟通架构，这些利益相关者包括设计人员、程序员、分析人员、项目经理、维护人员和支持人员。最后，**和利益相关者复审架构**这个任务使大家对架构达成一致意见。

如图 1.1 所示，这组与架构相关的工作产品随后流入了**创建逻辑详细设计**这个活动，在继续到**创建物理架构**这个活动之前，该活动把一些详细内容增加到了这个已确认的架构。

创建物理架构这个活动包含的任务与**创建逻辑架构**活动中的任务完全一样，只是**创建物理架构**时还需要考虑技术原因。因此，这些任务采用了承认这个区别的额外的最优方法。这组从**创建物理架构**活动中输出的与架构相关的工作产品流入了**创建物理详细设计**这个活动中，这个活动对已确认的架构元素增加了相应级别的细节，而这些细节可以用作实现的基础。实现最终产生一个可执行的软件版本，随后这个版本被测试以确保它达到与当前的迭代相适应的需求。来自架构回顾和测试活动的关于架构的反馈帮助指导下一次迭代的优先级和内容。

您已经了解了——那些最影响架构的任务方面的一个完整的迭代！在这些任务后面还有很多细节，本章我们只简要地描述一下它们。在本书的余下章节将详细说明这些任务和它们所基于的架构原理。

1.3　范围

本书关注软件密集型系统。因此，当没有限制时，架构、架构师和架构设计这些术语分别等同于软件架构、软件架构师和软件架构设计。虽然本书关注于软件密集系统，但是，在本书中还讨论了几个额外考虑的因素。例如，一个软件密集型系统仍需要硬件来执行，某些特性（如可靠性或性能）只有结合软件和硬件才能达到等，记住这些很重要。因此，不能忽视完整解决方案的概念，我们还会在随后的讨论中进行考虑。

您应该也意识到软件密集系统甚至支持一个更大的情景，在其中您可能会把由人和信息的组成系统看成是第一类公民，还有软件和硬件。一个更大的视角是考虑一个企业级架构，还应该提供业务流程蓝图和现有的 IT 系统，以及包括迁移计划及管理的业务改变优先权的视图。虽然我们承认这些观点以及它们在引导和约束我们作为软件架构师工作方面的影响，但企业级架构和系统工程不在本书的讨论范围内。

1.4 总结

本章重在为本书的余下章节的学习做好准备。在深入讨论之前，我们必须建立一些我们在余下章节会大量使用的核心概念。这些概念分别是架构师角色、他们执行的架构设计任务的特征，以及产生的架构。所有这些概念都将在第2章中进行阐述。

第 2 章

架构、架构师和架构设计

这一章概述了与本书主题相关的三个核心概念，最后对架构设计的好处进行了讨论。图 2.1 中所示的这些概念是**架构师**角色、架构师执行的**架构设计**任务的特点及由此生成的**架构**。

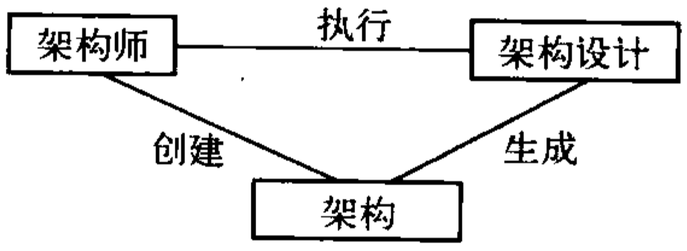


图 2.1　本书中使用的核心概念

2.1　架构

当提到架构的时候并不缺少架构的定义。甚至有一些网站还维护了这些定义集（SEI 2009）。本书中用到的定义来自于 IEEE 1471 2000，软件密集系统架构描述的 IEEE 操作规程建议（IEEE 1471 2000）。这个定义遵循（关键特性着重显示）：

> **架构**是体现在它的**组件**中的一个系统的基本**组织**、它们彼此的**关系**、与**环境**的关系及指导它的设计和发展的**原则**。（IEEE 1471 2000）

这个标准也详细说明了与这个定义相关的下列术语：

> **系统**是组织起来完成某一特定功能或一组功能的组件集。系统这个术语包括了单独的应用程序、传统意义上的系统、子系统、系统之系统、产品线、产品组、整个企业及感兴趣的其他集合。系统用于完成它的环境中的一个或多个**任务**。（IEEE 1471 2000）
>
> **环境**或上下文决定了对这个系统的开发、运作、政策以及会对系统造成其他影响的环境和设置。（IEEE 1471 2000）
>
> **任务**是由一个或多个**利益相关者**通过系统达到一些目标的系统的一个用途或操作。（IEEE 1471 2000）
>
> 系统**利益相关者**是对系统感兴趣的或与系统有关系的一个单独的团队或组织（或组织的一部分）。（IEEE 1471 2000）

正如您所见，在这个定义中用到了**组件**这个术语。然而，大多数的架构定义都没有定义**组件**这个术语，IEEE 1471 也不例外，故意让它比较含糊，因为这个术语在行业中存在很多解释。一个组件可能是逻辑上的或物理上的、独立于技术的或针对特定技术的、粗粒度的或细粒

度的。在本书中，我们使用统一建模语言（UML，Unified Modeling Language）规范中的**组件**的定义：

> 组件代表一个系统模块化的一部分，它封装了自身的内容，在环境中它的表现是可替换的。一个组件根据供给接口和需求接口定义自己的行为。同样地，一个组件也属于某一个类型，其一致性由这些供给接口和需求接口（包括它们的静态语义和动态语义）来定义。因此，一个组件可以由另一个组件替换，只要这两个组件类型一致。（UML 2.2 2009）

探讨**架构**的另一些定义以便能够观察这些定义之间的相似性，这是值得的。请看下面的这些定义，其中的一些关键特性已经突出显示了。

> 架构是关于下面这些内容的**重要的决策集合**：软件系统的组织、**构件**的选择及系统用于组装在一起的接口、这些构件之间相互协作的**行为**、把这些构件**合成**到日益变大的子系统、指导这个组织的**架构风格**——所有这些构件和它们的接口、它们的协作、它们的组合。（Kruchten 2000）

> 一个程序或计算系统的软件架构是一个或多个系统结构：它由软件要素、这些要素的外部可视属性和它们之间的关系组成。（Bass 2003）

> 一个系统或一个系统集的软件架构由所有关于组成这些系统的软件结构和这些结构之间交互的重要的设计决策组成。这些设计决策支持系统取得成功所需要的特性。这些决策为系统开发、支持和维护提供一个理论基础。（McGovern 2004）

您可以看到，虽然这些定义有所不同，但是，它们有很大的相似性。大部分定义都指出一个架构应该既关注结构又包括行为，应该关注重要的因素，遵循一种架构风格，受其利益相关者和环境的影响，基于基本逻辑使决策具体化。接下来的章节将讨论这些主题及其他内容。

2.1.1 架构定义结构

如果您要求某个人向您描述架构，十之八九，那个人会引用一些与结构相关的东西，通常是关于一个建筑或其他土木工程结构的，例如一座桥。虽然这些项目也存在其他一些特征，如行为、适用性，甚至美观性，但是，结构特性是最相似的，也最常提到。

所以，如果您要求某个人描述一个他正在处理的软件系统的架构，您可能会看到一张显示这个系统结构方面的图，无论是关于架构层、组件还是分布式节点方面的，这都不会令您感到惊讶。结构确实是一个架构的本质特征。

一个架构的结构状况会在许多方面描绘它们自己，而架构的大部分定义都是故意模糊的。一个结构部件可以是一个子系统、一个程序、一个库、一个数据库、一个计算节点、一个遗留系统、一个现有产品等。

架构的很多定义不仅承认结构部件本身，还承认结构部件的组合、它们的关系（和支持这些关系所需要的所有连接器）、它们的接口以及它们的分类。此外，其中每个部件都可以通过很多方法来提供。例如，一个连接器可以是一个套接字（socket），同步的或异步的，可以关

联某一特定协议等。

图 2.2 演示了结构部件的一个例子。这个图描述了在一个订单处理系统中含有一些结构部件的 UML 组件图。您可以看到三个组件，分别名为 Order Entry、Customer Management 和 Account Management。Order Entry 组件依赖 Customer Management 组件，还依赖 Account Management 组件，这些都通过一个 UML 依赖显示了出来。

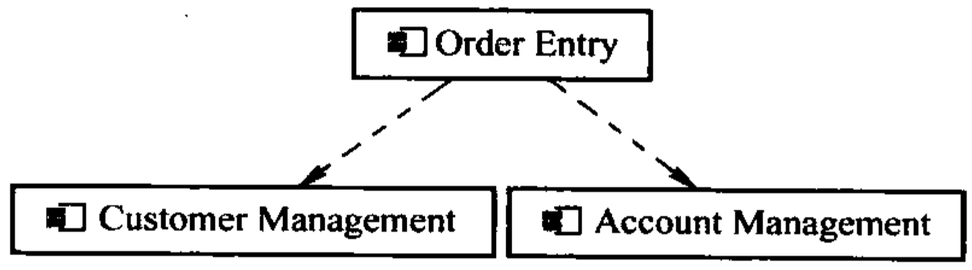


图 2.2 显示了结构部件的 UML 组件图

2.1.2 架构定义行为

架构除了定义结构部件之外，还定义这些结构部件之间的交互。这些交互提供了期望的系统行为。

图 2.3 是一个 UML 时序图，它显示了 5 个交互，合起来可以在一个订单处理系统中创建一个订单。首先，一个售货员（Sales Clerk）角色通过使用 Order Entry 组件的一个实例创建一个订单。这个 Order Entry 实例通过使用 Customer Management 组件的一个实例获取消费者详细信息。然后，这个 Order Entry 实例使用 Account Management 组件的一个实例创建订单，为这个订单组装订单项，然后下订单。

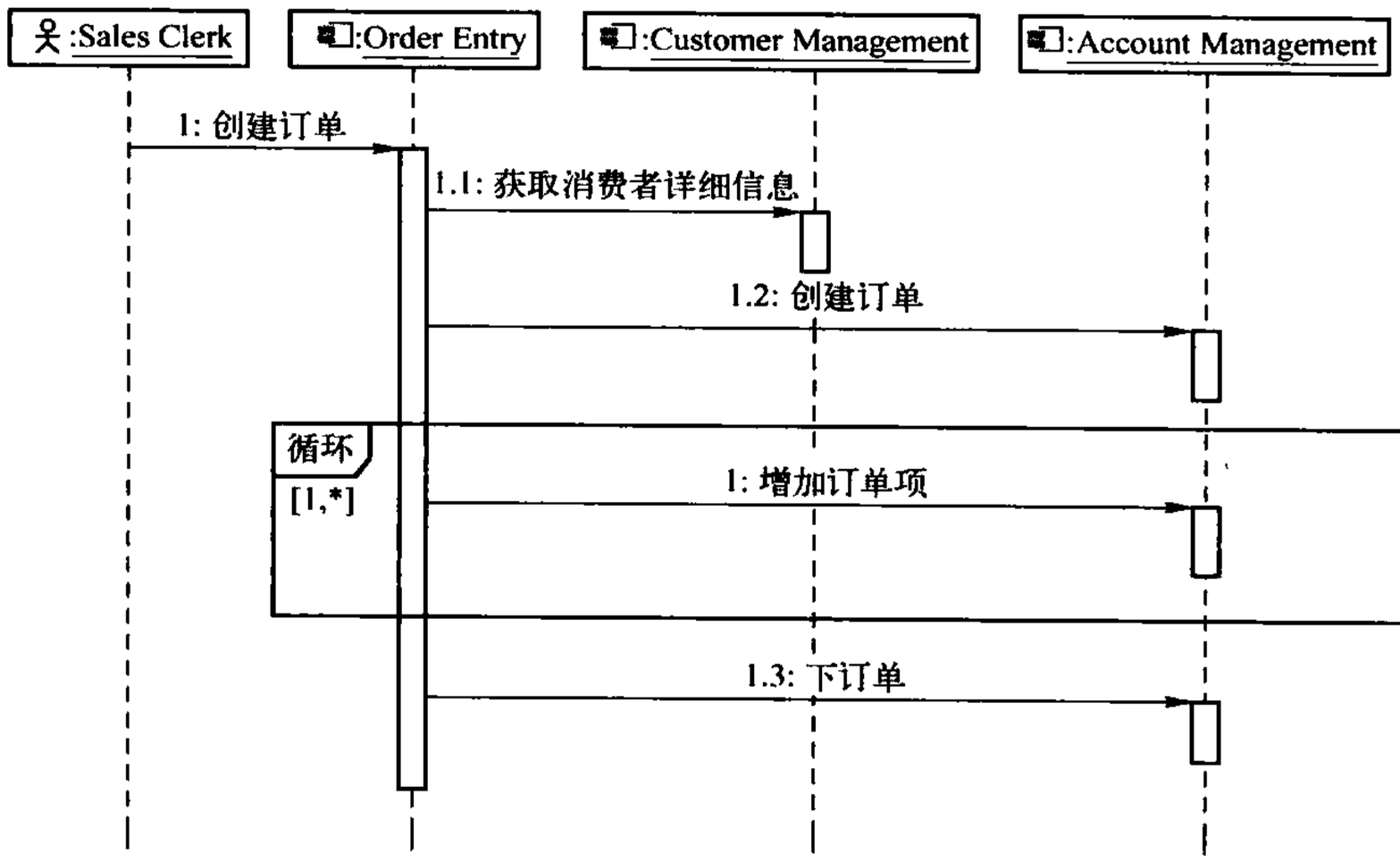


图 2.3 演示了行为元素的 UML 时序图

应当指出的是，图 2.3 与图 2.2 是一致的，因为您可以从图 2.3 中定义的交互推断出图 2.2 中显示的依赖。例如，一个 Order Entry 实例在执行过程中依赖 Customer Management 的一

个实例，如图 2.3 中的交互所示。这个依赖反射到相应的 Order Entry 和 Customer Management 组件之间的依赖关系，如图 2.2 所示。

2.1.3 架构关注重要的元素

虽然一个架构定义结构和行为，但是，它不关注定义所有的结构和行为。它只关注那些被认为是非常重要的元素。那些拥有长期和持久效果的是重要的元素，如那些主要的结构组件，那些与基本行为相关的元素，那些用来解决重要特性（如可靠性和可度量性）的元素。一般而言，架构并不关注这些元素的细节。架构的重要性也可以简单地表示为经济的重要性，因为认为某些元素优于其他元素的主要驱动力是创建的开销和改变的开销。

因为一个架构只关注重要的元素，它提供了考虑中系统的一个独特焦点——与架构师最相关的焦点。从这种意义上来说，架构是系统的帮助架构师管理复杂性的一个抽象。

值得一提的是，这个重要元素集不是固定不变的，它可能会随着时间发生改变。随着需求结果的提炼，风险的识别，可执行软件的构建以及经验教训的获取，这个重要元素集可能会发生改变。针对这些改变，架构的相对稳定性在某种程度上来说是一个优秀架构的象征，是架构设计流程执行良好的一个象征，是优秀架构师的一个象征。由于相对较小的改变，架构就连续地修改，这不是一个好征兆。然而，如果这种架构相当稳定，那这种架构就比较优秀。

2.1.4 架构平衡利益相关者的需要

一个架构的创建最终是为了解决一系列系统利益相关者的需求，但是，一般而言，满足所有的需求是不可能的。例如，一个利益相关者可能会要求在特定时间内获得某一功能，但是，这两个需求——功能和时间期限是互斥的。或者减小功能范围来赶上进度，或者延长时间来满足所有的功能。同样地，不同的利益相关者可能会提出相互冲突的需求，所以，必须再次达到一个适当的平衡。因此，进行折中是架构设计流程的一个基本方面，而协商是架构师的一个基本特性。

根据您手头的工作，考虑一组利益相关者的下列需求：

- 最终用户的需求和直观正确的行为、性能、可靠性、可用性、实用性及安全有关。
- 系统管理员的需求和直观的行为、管理及帮助监控的工具有关。
- 市场人员的需求和竞争特性、市场时机、与其他产品的定位及成本有关。
- 消费者的需求和成本、稳定性及进度有关。
- 开发人员的需求和明晰的需求、简单一致的设计方法有关。
- 相关经理的需求和项目追踪的可预见性、进度、资源产出率及预算有关。
- 维护人员的需求和易于理解、可靠及包含文档的设计方法有关，还与进行修改的容易程度有关。

正如您可以从这个清单中看到的那样，架构师的另一个挑战是利益相关者不仅仅关注系统提供必需的功能。列出来的许多关注点实际上是非功能性的（因为它们对系统的功能没有帮助）。这样的关注点代表着系统的质量或约束。就架构师关注的内容而言，非功能性需求通常

是最重要的需求；我们会在第7章“定义需求”中进行详细的讨论。

2.1.5 架构基于合理证据使决策具体化

架构的一个重要方面不仅是最终结果——架构本身，还要有解释为什么这样做的合理证据。因此，正如第4章“编写软件架构文档”所述，一个重要的考虑是导致这种架构的决策的文档证明以及这些决策的合理证据。

这些信息与许多利益相关者有关，尤其与那些必须维护系统的人有关。当架构师需要回顾他们所做决策背后的根本原因时，这些信息对他们自己也很有价值，他们可以不必重新回顾以前的步骤。例如，当架构师被检查并需要证明他们所做决策的正确性时，会用到这些信息。

还有，一些决策可能是强加给架构师的，从这种意义上来说，表现为对解决方案的约束。例如，可能存在使用特定技术和产品的公司层面的政策，而这个政策必须应用到解决方案中。

2.1.6 架构会遵循一种架构风格

大部分架构都源自于拥有相似关注点的系统。这种相似性可以称为是一种**架构风格**，这个术语借自于建筑架构中的风格这个术语。您可以把一种架构风格看成是一个特定的模式，虽然通常是一个复杂和复合的模式（好几个模式应用在一起）。

> 每个结构良好的软件密集型系统都充满了模式。（Booch 2009）

一种架构风格代表了一个经验规范，寻找机会重用这些经验规范对于架构师来说通常是一个良好的做法。架构风格的例子包括分布式风格、管道和过滤风格、数据集中的风格、基于规则的风格、面向服务的架构等。第5章中将进一步讨论架构风格。一个系统可能会存在不止一种架构风格。

> 一种**架构风格**根据结构组织的模式定义了一个系统家族。更明确地说，一种**架构风格**定义了一组组件词汇和连接器类型，以及一组如何把它们组合在一起的约束。（Shaw 1996）

架构风格（及一般的可重用资源）的应用可以使得架构师的生活稍微容易一点，因为这样的资源是已经验证过的，因此可以降低风险，当然也可以节省精力。一种风格在如何使用方面（以便可以较少地考虑如何去实现）以及在它的关键结构和行为方面（以便可以少编写一些架构文档，因为您可以简单地参考这个风格）通常都备有文档。我们将在第5章中详细地讨论架构资源。

2.1.7 架构受它的环境影响

一个系统存在于一个环境中，而这个环境会影响它的架构。有时候被称为环境中的架构。本质上，环境决定系统必须在其中运行的范围，进而影响架构。影响架构的环境因素包括架构支持的业务任务、系统利益相关者、内部的技术约束（如要求遵循组织标准）、外部的技术约束（如必须使用特定技术、与外部系统的接口或遵循外部的规定标准）等。

反过来，正如《软件架构实践第2版》（《Software Architecture in Practice，2nd ed.》，Bass 2003）中所述，架构也可能会影响它的环境。一个架构的创建可能会给拥有它的组织贡献可重用的资源，例如，因此使得这些资源可以被下一次开发使用。另一个例子是在架构内使用的软件包的选择，如一个客户关系系统（CRM，Customer Relationship Management），它随后会要求用户改变他们遵循的流程以适应这个包运作的方式。

2.1.8 架构影响开发团队的结构

一个架构定义了解决一组特定关注点所关联的要素集。例如，一个订单处理系统的架构将定义这些要素集：订单条目、账号管理、消费者管理、实行、与外部系统集成、持久和安全等。

其中每个要素集都可能要求一组不同的技能。因此，使软件开发团队的结构与它定义后的架构保持一致，这非常有意义。架构受最初团队的结构所影响，反之则不然。最好避免这个缺陷，因为结果通常会形成一个不完美的架构。正如康威法则所述，“如果您在一个编译器上有4个组在运作，您将得到一个4通编译器”。实际上，架构师们经常无心地创建影响创建这种架构的组织的架构。

虽然按符合架构的方式组织工作会比较有利，但是，这个有些理想化的观点并不总是具有可操作性。由于要完全注重实效，当前团队的结构和（当前团队的及维护团队的）可用的技能实际上就约束了什么才是可能，而架构师必须考虑这些约束。团队的地理分布是另一个必须考虑的约束。

> 架构的划分应该反映地理上的划分，反之亦然。架构上的职责应该进行分派，这样才可能在本地（地理上的）进行决策。（Coplien 2005）

2.1.9 所有系统都存在架构

还值得一提的是，每个系统都拥有一个架构，即使这种架构没有正式编写成文档，或这个系统极其简单，比如说只由一个元素组成。编写架构的文档有很大的价值，我们将在第4章中详细讨论这个主题。

无论是有意的或偶然的，基本上每个软件密集型系统都拥有一个架构。每个架构都以一种功能性的、经济的和优雅的方式控制了系统的能力。（Booch 2009）

如果没有为一个架构编写文档，要评定这种架构或证明它在灵活性、最佳方法的适应性等开发质量方面达到了所规定的需求是困难的（并非不可能）。随着时间的流逝，缺少文档也会使系统的维护变得极其困难。

2.1.10 架构有特定的范围

现在已经有许多种类的架构，最出名的是与建筑和其他土木工程结构相关的架构。即使在软件工程领域，您也会经常碰到不同形式的架构。例如，除了软件架构的概念之外，您可能会碰到像企业架构、系统架构、信息架构、硬件架构、应用架构、基础设施架构和数据架构这样

的概念。您还会听到其他的一些术语。其中任何一个术语都定义了架构活动的一个特定范围。

不幸的是，对于这些术语的意义或它们之间的关系，行业还没有达成一致，从而导致相同的术语具有不同的含义（同名异意）或同一件事存在多个术语（异名同义）。从图 2.4 中，你可以推断本书中使用的这些术语的范围，在本书中，我们关注软件密集型系统的架构。当您参考这张图以及随后的讨论时，几乎肯定会发现您不赞同的元素或在您的组织内以不同的方式使用的元素。通过这个例子可以认识到架构设计活动存在好几种可能的范围。

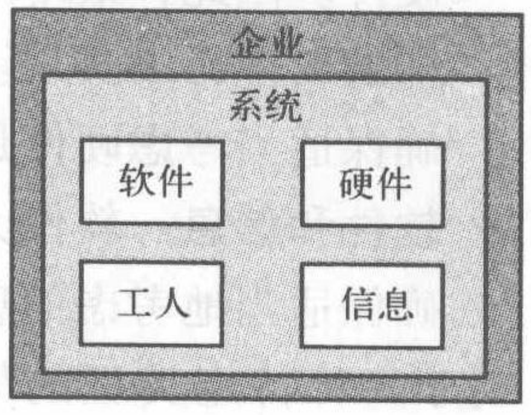


图 2.4　不同的架构设计范围

图 2.4 中元素的启示来自多个方面。IEEE Std 12207 1995（IEEE 信息技术标准 – 软件生命周期流程）如下定义一个系统：

> **系统**是由一个或多个流程、硬件、软件、设备和人所组成的、提供一些能力来满足特定的需要或目标的完整组合。（IEEE 12207 1995）

Rational 的系统工程统一流程（Rational Unified Process for Systems Engineering，RUP SE，也就是众所周知的模型驱动系统开发（Model-Driven Systems Development，MDSD））包含了一个类似的定义：

> **系统**是提供由一个企业用来实现业务目标或任务的服务的一组资源。系统部件通常包含硬件、软件、数据和工作人员。（Cantor 2003）

图 2.4 中列出的各种元素分别是：

- **软件**。这个元素是本书的主要关注点，考虑像组件、组件之间的关系和组件之间的交互这样的项目。
- **硬件**。这个元素考虑像 CPU、内存、硬盘、外围设备这样的项目。
- **信息**。这个元素考虑系统内使用的信息。
- **工人**。这个元素考虑系统与人相关的方面，如业务流程、组织结构、角色和职责、组织的核心能力。
- **系统**。正如先前的定义所示，一个系统包含软件、硬件、信息和工人。
- **企业**。这个元素类似于一个系统，因为它也考虑像硬件、软件、信息和软件这样的项目。然而，一个企业还可能跨多个系统，而且会对这些作为企业一部分的软件进行约束。与系统相比，企业与业务有更强的联系，因为企业关注于业务目标的达成，还关心像业务战略、业务灵活性及组织效率这样的项目。此外，企业还可能是跨公司的。

因为在本书中我们关注软件密集型系统，做一些补充说明是值得的。我们应该特别注意**软件密集型系统也是一个系统**。因此，您应该理解在软件和与其必须共存的元素之间的关系：

- **软件和工人**。虽然工人不在本书的考虑范围之内，但是，在系统必须提供的那些用来支持任何与系统进行交互的人的功能方面，肯定存在某一关系。在本书中，确保提供这些功能是应用架构师的职责。

- **软件和硬件**。软件密集型系统必须始终考虑的环境的一个特定方面是软件执行所在的硬件。因此，最终的系统是软件和硬件的一种结合，而这种结合使像可靠性及性能这样的属性得以实现。软件脱离了它执行所在的硬件就不能实现这些属性。在本书中，确保适当考虑硬件是基础设施架构师的职责。
- **软件和信息**。软件元素在它们的执行过程中可能会生成和消费持久信息。在本书中，确保适当地考虑信息是数据架构师的职责。

对于每种特定类型的架构，都存在一种相应类型的架构师（软件架构师、硬件架构师等），也存在相应类型的架构设计（软件架构设计、硬件架构设计等）。

既然您已经了解了这些定义，您可能有许多问题还没有找到答案。在一个企业架构和系统架构之间的区别是什么？一个企业是一个系统吗？在一些数据敏感的软件应用程序中，信息架构设计和数据架构设计相同吗？很不幸，这些问题没有一致的答案。目前，大家知道存在这些不同的术语，但是，行业对这些术语以及它们相互的联系还没有一致的定义。因此，我们建议您选择与您的组织相应的术语并适当地定义它们。然后，您会取得一定的一致性，至少，可以减少沟通中误解的可能性。

2.2 架构师

既然我们已经定义了通过架构我们所表达的含义，我们可以把注意力转移到负责创建架构的角色上：架构师。架构师这个角色在任何软件开发项目中都是最有挑战性的。架构师是项目中的技术领导，从技术角度来说，他最终承担项目的技术成功或失败的责任。

架构师是负责系统架构的人、团队或组织。（IEEE 1471 2000）

作为项目的技术领导，架构师必须拥有更广而不是更深的技能（虽然架构师们在特定领域应该拥有高深的技能）。

2.2.1 架构师是技术领导

首先，架构师是一位技术领导，这意味着架构师除了拥有专门技能外，还必须拥有领导能力。领导能力既体现在在组织中的职位上，也体现在架构师展现的品质上。

在组织中的职位方面，架构师（或首席架构师，如果架构师角色由一个团队来扮演）是项目中的技术领导，应该拥有进行技术决策的权威。另一方面，项目经理更关注于管理在资源、进度和成本方面的项目计划。用电影行业打一个比方，项目经理是制片人（确保事情完成），而架构师是导演（确保事情正确地完成）。根据他们的职位所导致的结果是，架构师和项目经理代表了这个项目的公共角色，作为一个团队，对于项目外的关注人员来说，他们是主要联系点。架构师尤其应该是创建一个架构及其给组织带来价值的投资的倡导者。

架构师也应该参与到决定如何组建团队中来，因为这种架构将意味着需要特定的技能。架构元素之间的依赖性会影响任务的先后顺序及这些技术所需要的时间，因此，架构师应该积极地为编制活动计划出力。需要提醒的是，因为架构师的成功与团队的素质紧密相关，参与面试

团队新成员也是架构师必要做的。

在架构师展现的品质方面，领导力也可以在与其他团队成员的交流中展现出来。特别地，架构师应该为他人树立榜样并在制定方向方面表示出自信。成功的架构师是以人为导向的，所有的架构师都花时间指导并培训他们团队的成员。这种方式有益于团队成员处理问题、有益于项目、最终有益于组织本身，因为某些最有价值的资源（人）变得更有技能。

另外，架构师必须非常关注交付的实际结果并必须作为项目在技术方面的驱动力。架构师必须能够进行决策（通常在压力之下）并确保这些决策被传达、理解并最终被执行。

2.2.2 架构师的角色可能由一个团队来履行

在角色和人之间是存在差异的。一个人可能会履行很多角色（玛丽是一个开发人员和一个测试人员），而一个角色可能由许多人履行（玛丽和约翰履行测试人员的角色）。由于架构师需要非常广泛的技能，所以，架构师角色通常由多个人履行。这种方式允许技能分布于几个人，每个人都充分运用他自己的经验。特别是，理解业务领域和各方面技术所必需的技能，往往由几个人才能很好地覆盖。然而，最终的团队必须平衡。

在本书中，**架构师**这个术语所指的是可以由个人或一个团队履行的角色。

> 一个团队是拥有共同目的、执行目标、拥有使他们可以相互负责的方法的、技能相互补充的小部分人。（Katzenbach 1993）

如果架构师角色由一个团队履行，拥有一个首席架构师非常重要，他具有先见之明，还是架构团队的单点协调人。没有这个协调人，架构团队的成员创造出内聚的架构，或作出决策是困难的。

对于一个不熟悉架构概念的团队来说，为了达成共同的目的、目标和步骤，建议团队应该创建并颁布一个团队规章。（Kruchten 1999）

优秀的架构师们知道他们的优势和弱势。无论架构师角色是否由一个团队来履行，架构师都有好几个可信的顾问支持。这样的架构师承认他们的弱点，并通过获得必须的技能或通过与其他人一起工作来弥补他们知识的缺陷进而弥补这些弱点。最优秀的架构通常由一个团队而不是个人创建，这仅仅因为当有多人参与进来时，见识更广和更深。

架构团队概念的一个缺陷（尤其在大的项目中）是，它可能被理解成一个象牙塔，它的产出仅仅为了显示智力而不是为了实用。通过确保和所有的利益相关者积极地商议、对架构及其价值进行讨论，以及确保考虑任何有影响的组织政策，这个误解可以从一开始就减到最小。

2.2.3 架构师理解软件开发流程

大部分架构师曾经都担任过开发人员，而且都非常了解定义并认可项目中使用的最佳实践的必要性。最明确的是，架构师应该了解软件开发流程，因为这个流程确保团队的所有成员都能协调地进行工作。

这种协调性可以通过定义涉及的角色、从事的任务、创建的工作产品、不同角色之间的

移交点来获得。因为在日常工作中架构师会影响许多团队成员，理解团队成员的角色和职责，理解他们正在生产和使用的东西，这对于架构师来说很重要。实质上，团队成员希望架构师能够指导他们如何完成他们的职责，架构师必须能够以和团队遵循的开发流程一致的方式作出反应。

2.2.4 架构师掌握业务领域的知识

除了掌握软件开发技术之外，还非常期望（有些人说是必须的）架构师理解业务领域，以便他们能够担任利益相关者和用户（他们理解业务）及开发团队成员（他们更熟悉技术）之间的中间人。

> **领域**是从事于某一行业的人理解的归纳为一组概念和术语的知识或活动范围。
> （UML User Guide 1999）

业务领域的知识也使得架构师更好地理解系统的需求，还能够确保捕获恰当的需求。另外，一个特定领域通常与能应用到这个解决方案中的特定架构模型组（和其他资源）相关，知道这个对照关系可以极大地帮助架构师。

因此，一个优秀的架构师通常平衡掌握软件开发知识和业务领域知识。当架构师理解软件开发但不理解业务模型时，可能会开发出一个不满足需求但反映这种架构师所熟悉内容的解决方案。

熟悉业务领域也使得架构师能够预见到他们架构中可能发生的改变。既然架构受其部署的环境（包括业务领域）影响很大，对业务领域的正确认识使得架构师能够在可能改变的区域和稳定性方面做出更全面的决策。举例来说，如果架构师认识到在将来的某点必须符合新的调整标准，他在架构中就会考虑这个需求。

2.2.5 架构师掌握技术知识

架构设计的某些方面明确需要技术知识，所以，一个架构师应该拥有一定程度的技术技能。然而，架构师不必是一个技术专家，他必须关注技术的重要因素，不是细节。架构师需要理解像 Java EE 或 .NET 这样的平台上可用的关键框架，但是不必理解访问这些平台可用的每个应用程序编程接口（API）的细节。由于技术变革相当频繁，架构师必须跟上这些变革。

2.2.6 架构师掌握设计技能

虽然架构设计没有局限于设计（正如您所见，架构师还参与需求任务），但是，很明显，设计是架构设计的核心方面。架构使关键设计决策具体化，因此，架构师应该拥有很强的设计技能。关键设计决策可以指关键结构设计决策、特定模型的选择、指导规格说明书等。为了保证系统的结构完整性，这些元素被代表性地广泛应用并对系统取得成功产生深远的影响。因此，这样的元素应该由拥有适当技能的人识别出来。

> 一个人不可能在短时间内获得设计的能力，而是多年经验累积的结果。甚至当设计专家回顾他们早期的工作时都惊讶原来的设计如此不好。当学习一项新技能时，为

了精通都必须进行设计实践。(Coplien 2005)

2.2.7 架构师具备编程技能

项目中的开发人员是架构师必须与其打交道的最重要的团队成员。毕竟，他们的工作产品最终交付生产用的可执行软件。只有当架构师承认开发人员工作的价值时，在架构师和开发人员之间的沟通才是有效的。因此，架构师应该具有一定的编程技能，即使他们在项目中不必编写代码，也必须跟上技术更新的脚步。

> 架构师应该有组织地参与开发并应该编写代码。如果架构师参与实现，开发组织会从架构师那儿获得见识，而那些见识可以直接有益于架构的专业知识本身。架构师还可以通过查看他们决策和设计的第一手结果进行学习，从而对开发流程给出反馈。(Coplien 2005)

大部分成功的软件架构师都曾经是核心的编程人员。某种程度上，他们就是通过这段经历了解到他们业务的某些情况。甚至当技术发展和引入新的编程语言时，优秀的架构师可以抽象出任何编程语言中的概念并应用这些知识对新的编程语言了解到必要的程度。没有这些知识，对于实现过程中架构上的重要元素（如源代码的组织、采用的编程标准），架构师将不能进行决策，在架构师和开发人员之间将会存在沟通障碍。

2.2.8 架构师是优秀的沟通人员

与架构师相关的所有软技能中，沟通最重要。有效的沟通涉及各个方面，架构师必须全部精通。尤其是，架构师应该拥有有效的口头、书面及表达技巧。同样，沟通应该是双向的。架构师应该既是优秀的聆听者，也是优秀的观察者。

有许多理由说明能够有效沟通是项目成功的基础。明显地，与利益相关者沟通对于理解他们的需求及与他们就架构达成（并保持）一致来说非常重要。与项目团队沟通尤其重要，因为架构师不是简单地负责把信息传达给团队，还要激发团队。特别地，架构师负责传达（并强调）系统的愿景，以便这个愿景为大家共享，而不是只有架构师理解并相信。

2.2.9 架构师进行决策

在有许多情况不清楚、没有充足的时间探究所有的可能性，以及存在交付压力的环境中不能进行决策的架构师不太可能继续工作下去。不幸的是，这样的环境很常见而不仅是例外，成功的架构师承认这个情形而不是设法改变它。即使架构师在决策时咨询其他人并营造其他人共同参与决策的一个环境，进行适当的决策仍然是架构师的职责，而这些决策并不总是正确的。因此，架构师必须是厚脸皮的，因为他们可能必须纠正他们的决策并原路返回。

没有进行决策的能力将慢慢地破坏项目。项目团队会对架构师失去信心，项目经理将会担心，因为这些等待架构师决策的事项没有进展。非常实际的危险是，如果架构师没有制定关于架构的决策并编写成文档，团队成员会开始制定他们自己的（可能不正确的）决策。

2.2.10 架构师知道组织政策

成功的架构师并不仅仅关心技术，他们还对政治敏感并知道在组织中的权力。他们利用这些知识确保与恰当的人沟通，并确保在项目的适当周期中获得支持。忽视组织政策是（相当）天真的。

> 政策包括大量的不确定性，这使许多技术人员紧张。这迫使他们在“客场”比赛，仿佛他们处于一个对他们不利的地方，因为他们的技术不能发挥多大的威力。（Marasco 2004）

事实是组织中起作用的许多强制约束位于项目交付的系统之外，而这些约束必须考虑。

> 人类的思想大都各不相同，为了解决不同的意见，一个政策性流程是不可避免的。因此，与其谴责它，倒不如把政策理解成处理不同意见这种必然需求的一种有效方式。（Marasco 2004）

2.2.11 架构师是谈判专家

对于架构设计的许多方面，架构师需要与许多利益相关者相互进行交流。其中的一些交流需要谈判技巧。架构师特别关注的一点是在项目中尽可能早地把风险减到最小，因为把风险减到最小对稳定架构所花的时间有直接的影响。因为风险与需求（及需求中的变化）有关，消除风险的一个途径是精炼需求以便这种风险不再出现，因此，必须回退需求以便利益相关者和架构师达成。这种情形要求架构师是一位有效的谈判专家，能够清晰明白地表明不同折中的后果。

2.3 架构设计

我们已经讲述了什么是架构，也已经定义了架构师这个角色的特点，下面我们来学习作为架构设计流程基础的内容或特点。我们不会深入讨论每个任务的细节，因为这些细节将在本书的后续部分讲到。另外，架构设计的优势将在这一章的后面讲到。

> 软件**架构设计**代表的是一个架构的定义、文档编写、维护、改进和验证正确实现的活动。（IEEE 1471 2000）

架构设计的范围相当广泛。图 2.5 显示了定义软件架构设计流程的不同方面的一个元模型。这个元模型来自于 IEEE 1471 标准，可以看作是架构师关注的架构设计不同方面的导航图。在本书中我们还将考虑元模型的另一些元素。例如，在第 4 章中（这一章我们将考虑如何编写一个架构的文档），我们将详细说明架构描述符（Architectural Description）元素。在附录 A 中，我们提供了本书中用到的全部元模型描述。

这个元模型中的关系直接取自于 IEEE 1471 标准，总结如下：

- 一个系统拥有一个架构。
- 一个系统实现一个或多个任务。

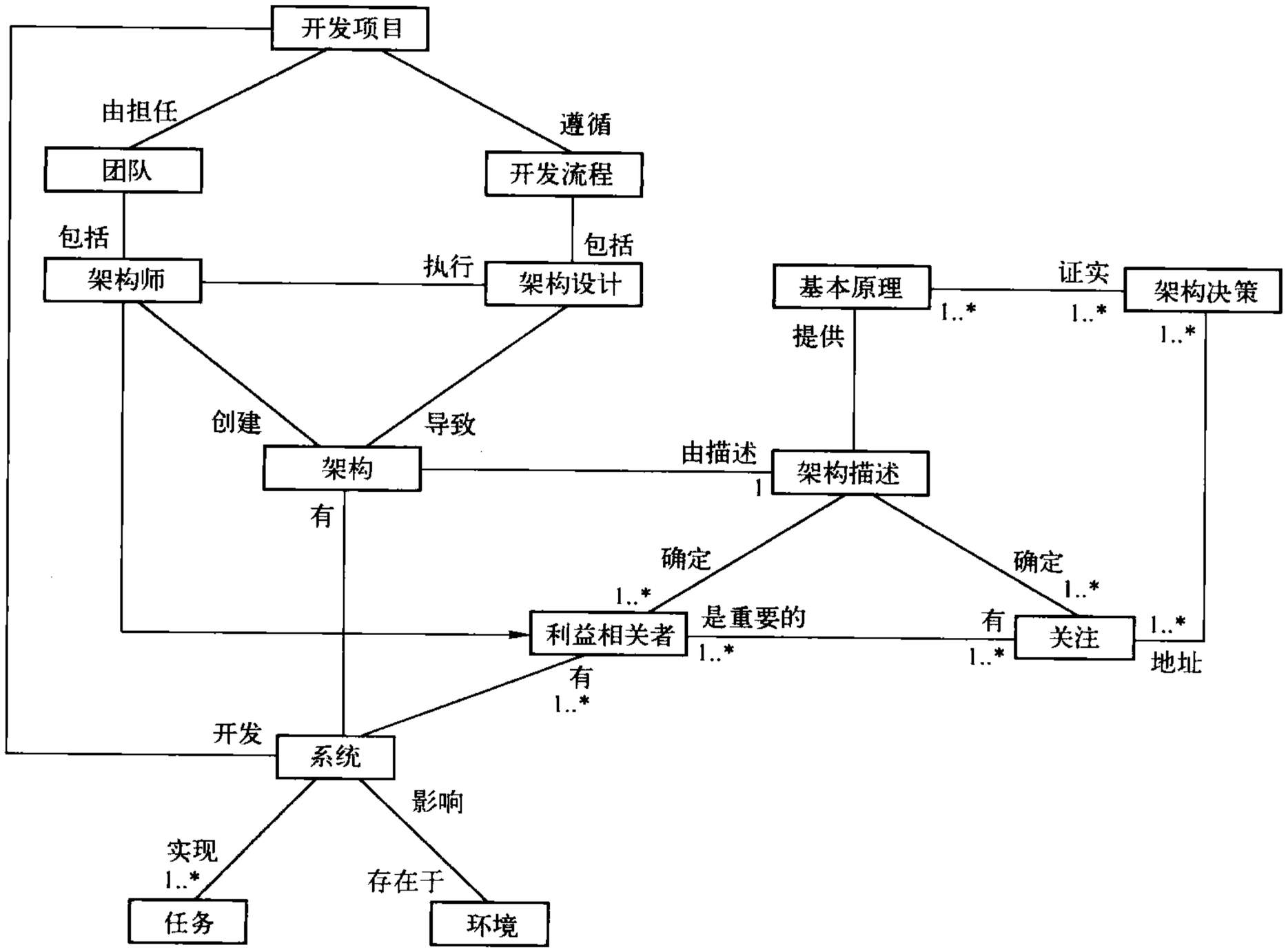


图 2.5　架构设计相关术语的一个元模型

- 一个系统拥有一个或多个利益相关者。
- 一个系统存在于一个环境。
- 一个环境影响一个系统。
- 一个架构由一个架构描述描述。
- 一个架构描述确定一个或多个利益相关者。
- 一个架构描述确定一个或多个关注点。
- 一个架构描述提供基本原理。
- 一个利益相关者拥有一个或多个关注点。
- 一个关注点对于一个或多个利益相关者来说很重要。

IEEE 1471 标准的一个附带好处是，它不仅适用于编写一个软件架构文档，还可以看作是架构师在工作中必须关注的概念推理框架。图 2.5 中不属于 IEEE 1471 标准的关系有：

- 一个开发项目由一个团队担任。
- 一个开发项目遵循一个开发流程。
- 一个开发项目开发一个系统。
- 开发流程包括架构设计。
- 团队包括一个架构师。

- 架构师执行架构设计。
- 架构师创建一个架构。
- 架构师是利益相关者的一种。
- 架构设计产生一个架构。
- 基本原理验证一个或多个架构决策。
- 一个架构决策处理一个或多个关注点。

2.3.1 架构设计是一门科学

架构设计是一门公认的学科，虽然它正在形成。它强调关注改进架构设计流程成熟度的技术、流程和资源。提升这种成熟度的一种方式是吸收现有的知识体。概括地说，架构师在开发一个架构时寻找已验证的解决方案而不是重复发明，从而避免不必要的创造。在参考架构、建筑和设计模式及其他可重用资源方面的系统化经验也有一定的作用。

然而，在软件架构设计流程像土木工程中的流程一样，成熟之前仍然有一段路要走。这种成熟度可以从许多方面进行考虑，包括标准的使用及对最佳方法、技术和流程的理解。

2.3.2 架构设计是一门艺术

虽然架构设计可以看作是一门科学，但总是需要提供一定程度的创造力，当处理新奇和没有先例的系统时尤其需要如此。在这些情况下，可以借鉴非系统化的经验。正如画家在面对空白画布时寻找灵感一样，架构师有时候也会把他们的工作看作是一门艺术而不仅是一门科学。然而，在很大程度上，架构设计的艺术性是很小的。即使在最新颖的系统中，通常也可以从其他地方复制解决方案，然后修改它，使其适应考虑中的系统。

随着软件架构设计流程变得越来越主流化，它似乎不再被看作是只有极少数人能掌握的神秘的方法，而仅仅看作是有一定科学基础并被公认的一组定义良好且经过证实的广泛接受的方法。

2.3.3 架构设计跨越很多方面

架构师参与软件开发流程中架构设计之外的许多方面：

- 架构师在需求方面提供帮助，例如，确保获取架构师特别感兴趣的那些需求。
- 架构师参与排定需求的优先级。
- 架构师参与实现，定义用于优化源代码及可执行工作产品的实现结构。
- 架构师参与测试，确保结构可测试并被测试。
- 架构师负责开发环境中定义一些项目标准及指导方针方面的一些工作。
- 架构师帮助定义配置管理策略，因为配置管理结构（支持版本控制）经常反映已经定义的架构。
- 架构师和项目经理紧密合作，架构师投入项目计划活动。

本书后面章节将深入讲述这些内容。

2.3.4 架构设计是一个渐进的活动

经验表明架构设计不是项目早期一次性执行的一项活动。比较确切地说，架构设计适用于项目的整个生命周期，架构随着一系列递增和迭代的可执行软件的交付而成熟。随着每次交付，架构变得越来越完善和稳定，这就引出了架构师在项目的生命周期中关注什么的问题。

成功的软件架构设计成果是以结果驱动的。因此，架构师的关注点会随着时间改变，因为期望的结果也随着时间在改变。图 2.6 中列出了这个概况，这个图由 Bran Selic 提供。

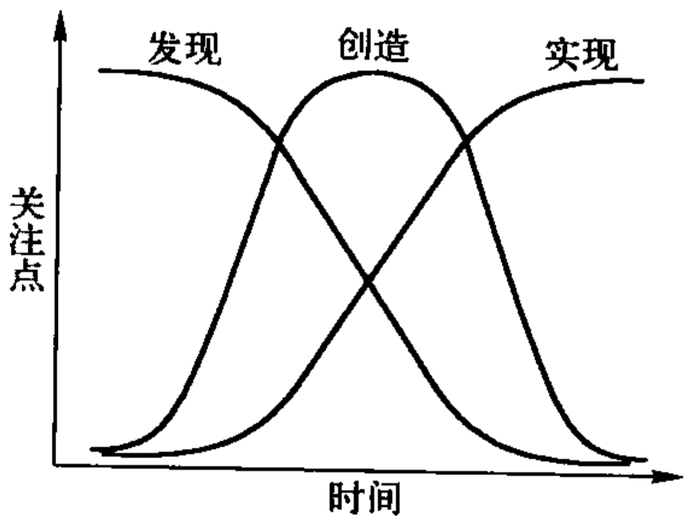


图 2.6 随着时间改变的项目重点

图 2.6 表明，在项目早期，架构师关注于发现。重点是理解系统的范围、识别重要的特征及任何相关的风险，这些因素明显影响架构。然后，关注重点改变为创造，架构师的主要关注点是开发一个可以为完全实现提供基础的稳定的架构。最后，当大部分发现和创造都已经发生后，关注重点改变为实现。

应该注意的是，关注于发现、创造和实现的顺序并不是严格不变的。在项目早期，当架构原型构建时就发生了一些实现，在项目后期，当接受一些教训后，也会进行一些发现，用于实现架构特定元素的不同策略会相应进行。在第 3 章中将更深入地讨论随时间改变的架构设计重点。

在系统交付之前架构设计的流程一直没有结束，因此，架构师必须参与到项目结束。一个组织经常强烈期望在架构稳定时把架构师从项目中移走以便把这个宝贵的资源用于其他项目。然而，在项目后期仍然需要进行架构的决策。实际上，经常可以发现一个妥协情况：在影响架构的主要决策制定之后，架构师成为团队的兼职人员。不管怎样，架构师不应该完全脱离。当架构师的角色由一个团队来担任时，情况就会灵活得多，因为其中部分成员可以用于其他项目，只要留下来的成员继续确保系统的架构完整性即可。

2.3.5 架构设计受许多利益相关者驱动

一个架构实现许多利益相关者的需要。因此，架构设计的流程必须考虑所有的利益相关者，以确保他们的关注点（尤其那些可能对架构有影响的关注点）都被捕获、阐明、协调及管理。使相关的利益相关者参与对这些关注点的解决方案的复审也很有必要。

考虑所有的利益相关者对于确保生成一个成功的架构非常重要。这些利益相关者影响架构设计流程的许多方面，正如本书将深入讨论的，包括需求的采集方式、架构文档编写方式及架构的评定方式。

2.3.6 架构设计经常包括折中

既然有这么多因素影响一个架构，很明显架构设计的流程包括进行折中。经常需要在冲突的需求之间进行折中，利益相关者必须商议来帮助进行正确的折中。需要折中的一个例子是在

成本和性能之间，对一个问题投入更多的处理能力将改善性能，但会提高成本。下面可能是一个需求方面的冲突，假定架构师一直忙于探索所有需求的可能性，但必须与那些需求冲突的利益相关者进行沟通。

在解决方案领域存在另一种折中。使用这一种技术还是另一种技术，使用这一个第三方组件还是另一个第三方组件，乃至使用这一组模式还是另一组模式，所有这些折中都与解决方案而不是需求相关。进行折中并不是架构师能够或应该避免的事情。架构师被期望有选择地考虑，在其中进行折中是架构设计流程的基本方面。

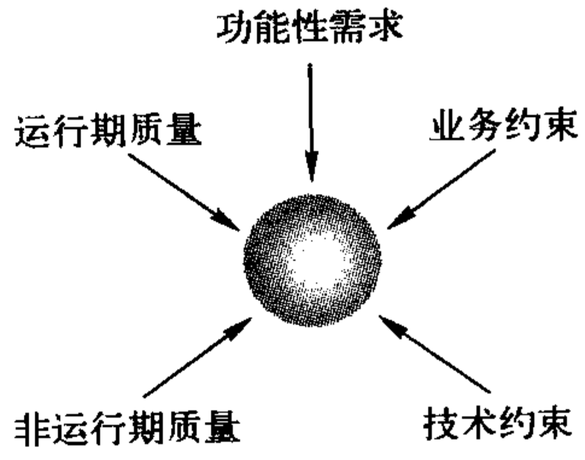


图2.7 用折中解决对立的约束

图2.7提供了解决方案架构设计的一些影响因素的简单分类。除了系统提供的功能外，您必须关注非功能性需求，包括运行期质量（如性能和可用性）、非运行期质量（如可维护性和便携性）、业务约束（如规章制度约束和资源约束）及技术约束（如委托的技术标准和委托的解决方案组件）。

2.3.7 架构设计承认经验

架构师很少从一张白纸开始做起。正如先前所述，他们积极地寻找那些可能编成架构模式的经验、设计模式、现有组件等。换句话说，架构师寻找可重用的资源。只有最无知的架构师才不考虑经验。

可重用资源是对重复出现问题的解决方案。可重用资源是牢记在头脑中的已经开发过的资源。（RAS 2004）

当然一个架构的元素在当前系统的上下文中可以重用，但是，架构师们还把他们的架构或架构的元素看作是可以在当前系统之外可以重用的资源。在这一章后面及第5章“可重用架构资源”中将讨论重用的主题。

2.3.8 架构设计既由上而下也由下而上

许多架构通常采用由上至下的设计方式，在这种方式中，先是获取利益相关者的需求，在架构定义之前了解需求，然后设计架构元素，最后实现这些元素。然而，架构很少完全由上至下驱动。主要原因是大部分系统并不是完全从头开始的。通常有一些遗留产品以必须采纳且影响这种架构的现有解决方案元素的形式存在。这些元素的范围从完全重新开发的应用到约束这种架构的委托设计或委托实现元素。举例来说就是设计成使用一个关系型数据库的约束，还是设计成和一个现有系统交互这样的约束。

当从任何已经创建的可执行软件中吸取教训时，架构也会由下至上驱动，如一个架构概念模型，从中获得的教训导致架构相应变得精炼。

成功的架构师承认这两种架构设计的方式都是必需的，他们的架构既由上而下也由下而

上，可以看作是架构设计“中间相会”的方式。

2.4 架构设计的优点

总的来说，架构设计是降低成本、改进质量、支持按计划及时交付、支持按需求交付及降低风险的一个关键因素。在这一部分，我们集中讲述架构设计有助于达成这些目标的更明确的优点。

另外，因为架构师有时候必须证明架构设计的存在是合理的，这一部分将提供一些有用的资料来证明架构设计是软件开发流程中的一个重要组成部分。

2.4.1 架构设计解决系统的质量问题

系统的功能性是通过组成架构的各种元素之间发生的交互来支持的。不管怎样，架构设计的关键特征之一是系统质量是通过架构这个工具达到的。在缺少统一的架构设想的情况下，像性能、安全及可维护性这样的质量是不可能保证的，这些质量不仅限于单个架构元素，而是遍布于整个架构。

例如，为了处理性能的需求，必须考虑架构的每个组件执行的时间以及组件之间通信所花的时间。同样，为了处理安全的需求，必须考虑组件之间的通信类型，而且在必要的地方引入专门的具有安全意识的组件。所有这些考虑都是关于架构的，而且，在这些例子中，涉及这些组件本身以及它们之间的联系。

架构设计的一个相关优点是使得在项目生命周期的早期评估这类质量成为可能。通常特意创建架构的概念模型以确保处理这类质量。通过一个真实的实现（在这种情况下采用架构概念模型）来证明达到这类质量要求，这很重要，因为一个架构无论在纸上看起来多么优秀，只有可执行软件才是这种架构已经处理这类质量的唯一真实标准。

2.4.2 架构设计促进达成共识

架构设计流程促进不同的利益相关者达成共识，因为它提供了一个工具使大家能够对系统解决方案进行辩论。为了支持这样的辩论，架构设计流程必须确保架构被清楚地交流和证实。

一个可以被有效交流的架构使得大家对决策和折中进行辩论，使检查变得容易，还使得大家达成一致。相反，一个缺乏交流的架构不会使这样的辩论发生。没有这样的输入，最终的架构可能是低质量的。很明显，对一个架构进行有效交流的一个重要方面是为它适当地编写文档。这是架构师的主要关注点，也是第 4 章“编写软件架构文档”的主题。

需要注意的是，架构可以作为培训的一部分，促进架构师（及他们的设想）和新的或现有的团队成员之间达成共识。为了取得这个好处，架构还必须进行有效地交流。清楚知道他们正在实现的内容的开发团队会更有可能按期望实现产品。

通过验证架构是否符合规定的需求也可以促进达成共识。正如前一部分提到的，创建可执行概念模型是证明一个架构达到一定运行期质量的一个极好方式。

2.4.3 架构设计支持计划编制流程

架构设计流程支持好几个方面。很明显，它支持详细设计和实现的活动，因为架构直接就是这些活动的输入。然而，在架构设计流程带来的好处方面，主要的好处是为项目计划编制和项目管理活动（一般指时间安排、工作分配、成本分析、风险管理及技能发展）提供了相关支持。架构设计流程可以支持这些方面，是架构师和项目经理应该保持紧密关系的主要理由之一。

其中的许多支持来自于这个事实：架构确定了系统中重要的组件及这些组件相互的关系。考虑图2.8中的UML组件图，为了便于讨论，它被特意简化了。这张图显示了4个组件及它们之间的依赖关系。

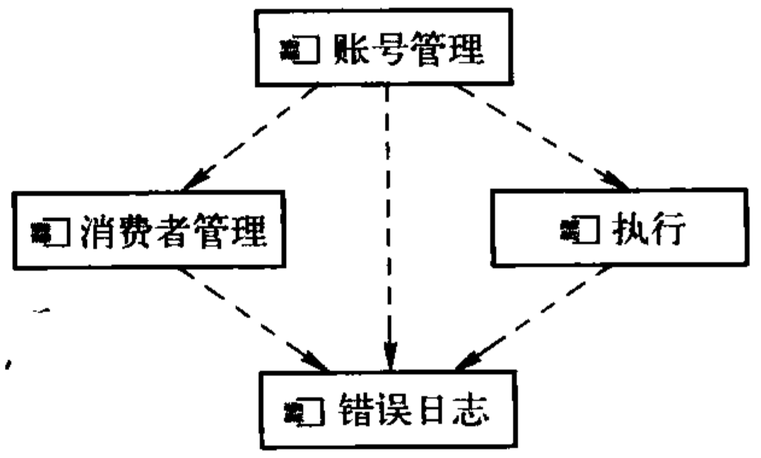


图2.8 显示架构重要元素的UML组件图

为了便于讨论，我们考虑一个简单的情形，每个组件总是完全实现的（就是说，我们不会创建每个元素的部分实现，也不存在与实现分离的接口）。在时间安排方面，这些依赖表明在一个订单中所有元素都应该考虑。例如，从实现的角度来看，这些依赖告诉您“错误日志”组件应该在一开始就必须实现，因为其他组件都使用了这个组件。接下来，“消费者管理”及“执行”组件可以并行实现，因为它们彼此并不依赖。最后，当这两个组件实现时，就可以实现“账号管理”组件。根据这些信息，您可以得到如图2.9所示的一张甘特图（项目经理使用的计划编制的常规技术之一）。每个任务所需的时间确实需要考虑，但是，可以根据每个架构元素的复杂度部分地推断出来。

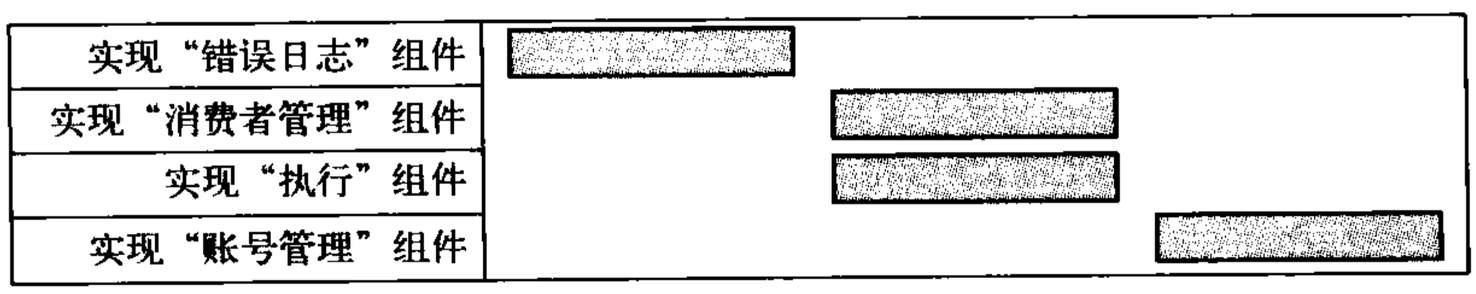


图2.9 基于架构重要元素之间的依赖形成的甘特图

在项目的成本评估中架构师也能提供帮助。与一个项目相关的成本来自许多方面。很明显，任务的周期和分配给任务的资源可以确定劳动力的成本。架构也可以帮助确定交付系统中使用的第三方组件相关的成本。另一个成本来自于支持创建架构元素所需使用的特定工具。架构设计也涉及区分风险高低级别和确定适当的风险规避策略，这两者都是项目经理的输入。

最后，架构确定可以在项目需要的技能方面提供输入的解决方案分立组件。如果在项目内或组织内没有适当技能的资源可用，架构可以清晰地帮助识别技能需求的地方。这种需求可以通过发展现有的人员、外包或雇用新人员来得到满足。

2.4.4 架构设计促进架构的完整性

架构设计流程的主要目标之一是确保这种架构为设计人员和实现人员所做的工作提供一个稳固的框架。很明显，这个目标并不是简单地传播一个架构设想。为了确保最终架构的完整性，架构师必须清楚地定义架构本身，确定架构上重要的元素，如系统的组件、它们的接口和相互之间的交互。

架构师还必须为设计人员和实现人员定义在工作中指导他们的适当的方法、标准和指南。架构设计的另一个目标是消除设计人员及实现人员不必要的创造力，通过对设计人员和实现人员能做的工作进行约束来实现这个目标，因为违反这些约束会导致架构损坏。帮助确保架构完整性的另一个方面是采取一些确认设计人员和实现人员坚持架构标准和指南的适当的检查和评估活动。

我们将在第 8 章和第 9 章中进一步讨论架构评估。

2.4.5 架构设计有助于管理复杂性

当今的系统比以往的更加复杂，而这个复杂性必须进行管理。正如这一章先前提到的，因为一个架构只关注重要的元素，它提供了系统的一个抽象概念，从而为管理复杂性提供了一个方法。此外，架构设计流程还考虑组件的递归分解，这明显是碰到一个大问题后把它分解成一系列较小问题的好办法。

最后，管理复杂性的另一方面是使用容许架构的抽象概念进行交流的技术。例如，您可能会选择把组件分组成子系统或把实现与接口相互分离。如今，为了编写软件密集型系统的架构文档，采用表达抽象概念的行业标准（如 UML）非常常见。

2.4.6 架构设计为重用提供基础

架构设计流程既可以支持生产也可以支持可重用资源的消费。可重用资源对组织有益，因为他们可以降低一个系统的总成本，也可以提高系统的质量，假定可重用资源是已经被证实的（因为它已经被用过了）。

在资源消费方面，架构的创造支持了可能的可重用机会的识别。架构上的重要组件、它们相关的接口及质量的识别支持了非定制组件、现有系统、预装应用程序等可以用于实现这些组件的选择。

在资源生产方面，架构可能包含一些在当前系统之外也可以（自然）使用的元素。一个架构可能包含一个可以在其他背景中也能使用的错误日志机制。这样的重用通常是碰运气的，然而，一个战略性的重用会预先主动地考虑可用的资源。我们将在第 10 章中略微谈一下这个主题。

2.4.7 架构设计降低维护成本

架构设计流程在多个方面能够帮助降低维护成本。首先，架构设计流程应该始终确保系统

维护人员是一个关键利益相关者，维护人员的需求会作为主要关注点进行处理，而不是事后考虑。结果应该是形成一个适当编写了文档从而使系统的维护变得很容易的架构；架构师也确保系统维护机制的一致性并在创建架构时考虑系统的适应性及可扩展性。另外，架构师还会考虑维护系统所需要的技能，这些技能可能会与创建这个系统的团队成员的技能不同。

架构师应该考虑系统最可能发生改变的区域并隔离它们。如果改变只影响单个组件或少量的组件，那这个流程会相当简单。然而，有一些改变不能按这种方式隔离，例如与系统质量相关的改变（如性能或可靠性）。由于这个原因，架构师在设计当前系统的架构时必须考虑将来任何可能的需求。例如，把一个最初设计成支持几十个用户的系统放大到支持数千个用户，在不从根本上改变架构的情况下这几乎是不可能的。

可维护性问题仅仅是那些会随着时间逐渐发展的系统所主要关注的问题，并不针对那些打算提供小规模解决方案且生命周期短暂的系统。

2.4.8 架构设计支持影响分析

架构设计的一个重要优点是使架构师能够在进行调整之前推论出该调整的影响。一个架构确认出主要的组件、它们的交互、组件之间的依赖及组件到它们实现的需求之间的可追溯性。

根据这些信息，一个需求的改变对协作实现这个需求的组件的影响都可以分析出来。同样地，改变一个组件对于依赖它的其他组件所造成的影响也可以分析出来。这样的分析非常有助于确定一个改变的成本、改变对系统的影响及其带来的风险。

2.5 总结

这一章定义并解释了本书中使用的核心概念：架构、架构师和架构设计。这一章也讨论了软件开发流程采用以架构为中心的方式所带来的好处。然而，仍然有许多问题没有解决。例如，实际上架构师在软件开发项目中做什么、架构师产出什么、架构师角色如何与项目的其他角色发生联系。

定义了这些核心概念后，在第3章中我们把注意力转向这些概念在整个软件开发流程中的应用。

第3章

方法基本原理

讨论了一些核心概念之后，现在让我们把注意力转向支持架构设计流程的一些详细信息，如工作产品、任务和角色。为了提供这个概要，我们考虑了与软件行业中已经开发的最优方法相关的各种架构，并提取了各种方法的原理，包括 Rational 统一过程（Rational Unified Process）、IBM 统一方法框架（Unified Method Framework）、OpenUP、极限编程（XP）、Scrum、特征驱动开发（FDD，Feature-Driven Development）和精益方法。我们还考虑了相关的标准化规范，如软件系统流程工程元模型规范（SPEM，Software and Systems Process Engineering Metamodel Specification）。

这一章的目的是为了提供本书所基于的关键方法元素的纵览。因此，这一章是为后续几章作准备，我们将在后续章节中详细说明这些概念。附录 C 也列出了本书中讲述的方法元素的概要。

3.1 关键概念

为了使这一章中的后续讨论有意义，确定一个方法中提出的一些基本概念对我们来说很重要。软件系统流程工程元模型规范（SPEM）对象管理组（OMG）标准（SPEM 2007）可以帮忙，因为它提供了这些概念的定义，在本书中我们将使用这些定义。SPEM 标准支持并受好几个现有软件开发方法的影响，包括 OpenUP（OpenUP 2008）、Rational 统一过程（RUP 2008）、IBM 统一方法框架、富士通 DMR Macroscope 和 Unisys QuadCycle。

SPEM 标准非常详细地定义了各种概念。然而，在本书中，我们采用了部分简单的定义并进行了一些简单化的假设。本书中用到的相关术语如图 3.1 所示，由谁使用，在哪儿定义，关系及图标均来自 SPEM 标准。

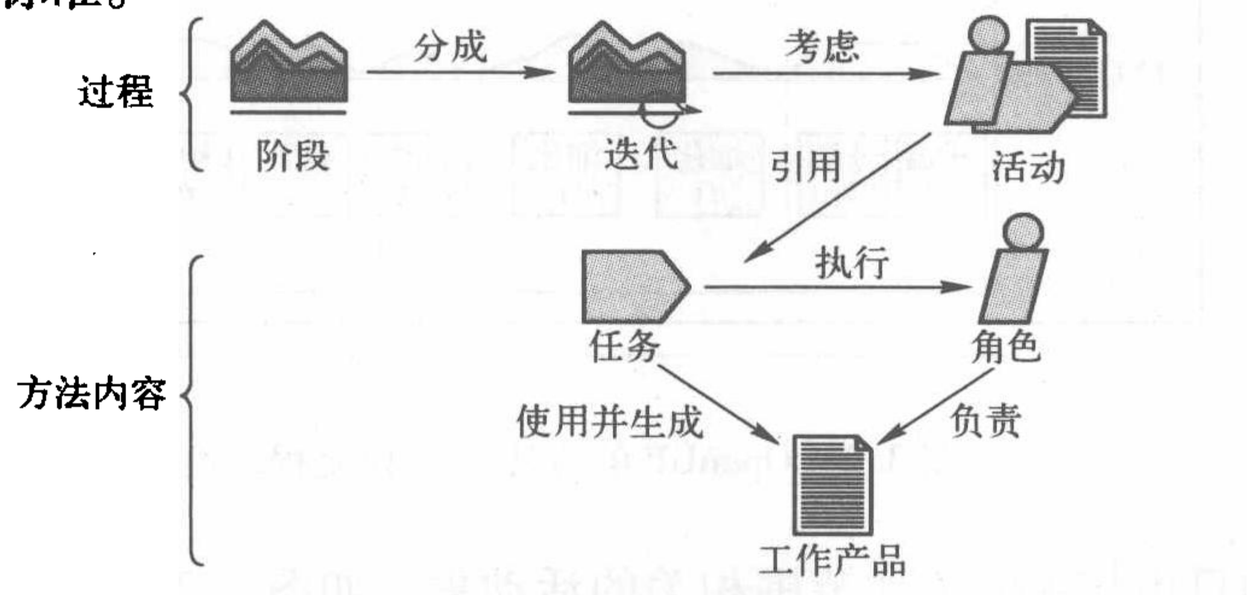


图 3.1　关键方法的概念及它们的关系

基本上，一个有效的软件开发方法应该描述由谁做什么、如何做以及什么时候做。本书就是按照下面的这些关键概念介绍的。

- 角色：谁
- 工作产品：做什么
- 任务：如何做
- 阶段、迭代及活动：什么时候做

另外，一个方法以模板、样例和技巧的形式提供指导。参考图 3.1，一个软件开发项目通常经历好几个**阶段**，每个阶段又划分为好几个**迭代**（虽然，正如您看到的，不是所有的流程都是按阶段和迭代来组织的）。在每个迭代中，我们考虑各种**活动**及它们涉及的**任务**，执行它们会获得一个特定的结果。任务由适当的**角色**执行，角色会使用并生成相关的**工作产品**。

如图 3.1 所示，也可以把一个方法看作由方法内容和流程组成。方法内容描述不受生命周期约束的元素，如角色、任务和工作产品。然后，一个流程取得这些元素并定义一个应用它们的顺序，根据项目的种类，还考虑像阶段、迭代和活动这样的概念。图 3.2 中提供了一个方法内容和流程分离的例子，它是通过 OpenUP 绘制的。在这个图中，竖轴表示方法内容，按专业（discipline）分组，而横轴表示流程。

专业是把定义一个主要“关注范围”和/或协作工作成果的任务组织起来的一个主要分类机制。

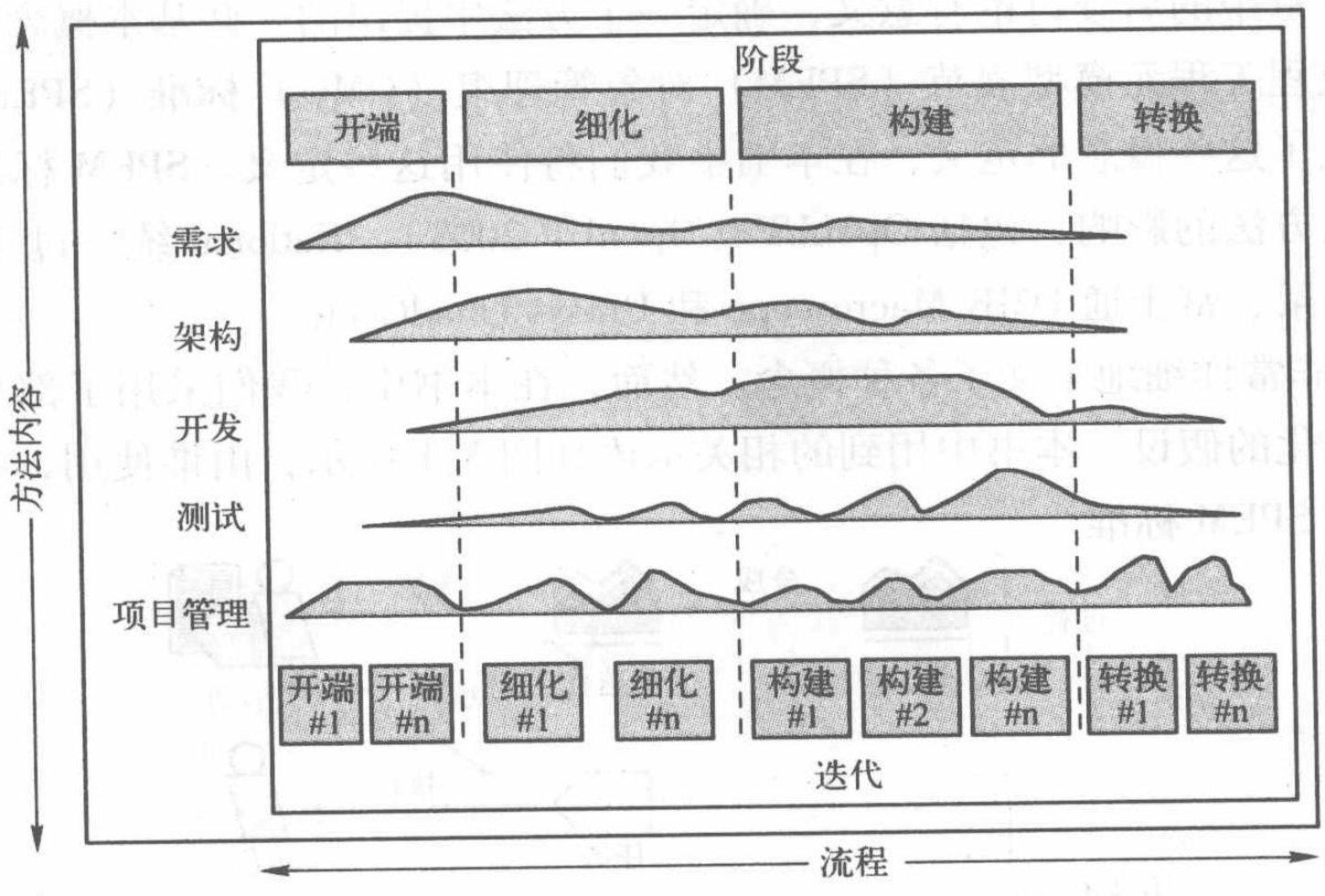


图 3.2 OpenUP 的方法内容和流程维度

专业是整个项目中与主要关注范围相关的活动集。如图 3.2 所示，OpenUP 围绕着 5 个专业进行组织，如表 3.1 所示。

表 3.1 OpenUP 专业摘要

OpenUP 专业	描　述
需求	这个专业解释了如何得出、分析、详细说明、验证和管理一个系统要开发的需求
架构	这个专业解释了如何根据架构上重要的需求创建软件架构。这种架构在开发专业中构建
开发	这个专业解释了如何设计和实现一个符合架构且支持需求的技术解决方案
测试	这个专业解释了如何通过设计、实现、运行和评估测试，提供关于这个正在完善的系统的反馈
项目管理	这个专业解释了如何指导、促进和支持团队，帮助它处理构建软件时的风险和障碍

正如图 3.2 中的“驼峰”所示；相对的专业重点会随着项目的阶段发生改变。例如，在早期的迭代中，较多的时间花在需求上，在随后的迭代中，较多的时间花在开发上。

角色、任务和工作产品的定义通常认为具有很高的可重用性，因为不像流程相关的元素，它们在不同类型的项目和生命周期中不会发生很大的变化。重新开发一个系统的软件开发项目与改变一个现有系统的项目会采用同样的步骤执行一个任务，如识别功能性需求，但是，在整个生命周期中，这个任务的重点差别很大。

3.2 方法内容

正如我们在这一章前面提到的，方法内容引用了组成方法的角色、工作产品和任务。下面对其进行详细介绍。

3.2.1 角色

角色定义了软件开发组织内作为团队一起工作的一组人或单个人的职责。角色负责一个或多个工作产品并执行一组任务。例如，业务分析人员这个角色负责**功能性需求**工作产品并执行**识别功能性需求**这个任务。在本书中，当一个角色执行一个任务时，这个角色会看作是主要角色或次要角色，正如在补充内容“概念：主要角色和次要角色”中论述的。

需要强调的是，角色不是单个人。单个人可能会扮演多个角色（拥有多个头衔），而多个人可能会扮演同一个角色。**项目经理**在编制项目计划和安排项目人员时负责把人员与角色对应起来，当然，**项目经理**在做这件事的时候还要考虑其他问题。本书中使用的与架构师相关的角色如图 3.3 所示（虽然不是所有的项目都需要这些专门的角色，有一个**架构师**角色就足够了）。

图 3.3 与软件架构相关的角色

首席架构师全面负责定义系统架构的主要技术决策。这个角色也负责为这些决策提供理论

基础；平衡各种利益相关者的关注点；管理技术风险和问题；还确保决策被有效地沟通、验证和执行。

应用架构师关注那些使业务流程自动化和满足业务需要的元素。这个角色主要关注业务需要的功能，但是，他也关心应用相关的元素如何满足系统的非功能性需求（质量和约束）。

基础设施架构师关注那些不依赖业务功能的系统元素，如持久机制、硬件和中间件。这样的元素支持应用相关元素的执行。这个角色关注那些对系统质量有明显影响的元素，因此，他也会处理一些延伸的非功能性需求。

数据架构师关注系统的数据元素，尤其是那些用适当的机制（如数据库、文件系统、内容管理系统或其他存储机制）保持持久的数据。这个角色定义适当的与数据相关的属性，如结构、来源、位置、完整性、可用性、性能和使用年限。

> **概念：主要角色和次要角色**
>
> 一个任务的主要角色被认为是对该任务负责并不可缺少的。次要角色被认为是有益于该任务，但并不对其负责，是可以缺少的。
>
> RACI 分类给出了一个比较完整的角色特征，其中，每个角色都可能对任务负责（Responsible），对任务有责任（Accountable），商议（Consulted）（征询角色扮演者的意见）和告知（Informed）（告知角色扮演者任务的进展和它的内容）。我们简化的分类把主要角色看作是对任务既负责又有责任的角色，把次要角色看作是商议和告知的角色。

3.2.2 工作产品

工作产品是在流程执行过程中生成和使用的一些信息或物理实体。工作产品的例子包括模型、计划、代码、可执行代码、文档、数据库等。虽然可能有多个角色协作来生成一个工作产品，但是它是单个角色的职责。角色把工作产品作为任务的输入，生成或修改工作产品作为他们执行任务的输出。架构师参与生成和使用工作产品的各种任务，正如这一章后面及与案例学习相关的几章中讨论的那样（第6~9章）。

SPEM 定义三种类型的工作产品：工件（artifact）、可交付物及成果。**工件**是一个实在的工作产品，如一个文档、模型、源代码、可执行物或项目计划。**可交付物**是切实的打包交付的工作产品。可交付物用于表示对一个利益相关者有价值（物质或非物质的）的一个工作产品。**成果**表示一个结果或任务执行结果的状态。与工件和可交付物不同，成果不代表潜在的可重用资源。

本书中讲述的方法主要集中在工件上。除工件外，架构师为**软件架构文档**（Software Architecture Document）可交付负责。架构师明确负责的工作产品如图 3.4 所示，其中还列出了不同的架构师角色。任意特定的方法都可以增加或删除图中总结的一些工作产品。

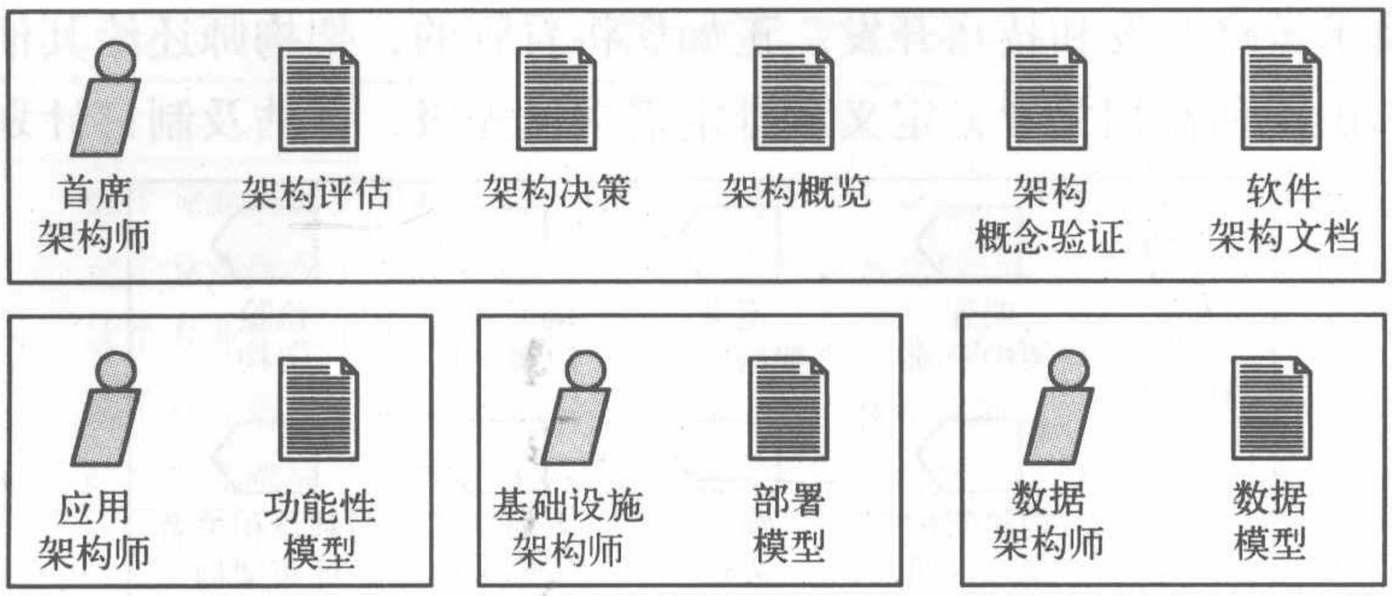


图 3.4 架构相关角色拥有的工作产品

这张图可能会产生两个令人误解的印象。第一个印象是所有的工作产品都是文档（因为工作产品的 SPEM 图标看起来像是一个文档），其实并不是。例如，**功能性模型**和**部署模型**可能会通过一个适当的建模工具表述成 UML 模型，而**架构决策**工作产品可能通过一个项目的 wiki 获取。第二个印象是与每个工作产品的创建相关的仪式级别非常依赖环境（如系统的复杂性、生命周期中所处的时间点等）。例如，**架构概览**工作产品可能只有一页纸，而**架构决策**工作产品由电子表格中的三项组成。

虽然图 3.4 中没有列出来，但是还有一些需要架构师帮忙但并不属于架构师的工作产品，例如**排定优先级的需求清单**。另一个例子是 **RAID**（风险、假设、问题和依赖条件）**日志**工作产品，架构师对其作出贡献，但它属于项目经理。正如您所想，在负责一个任务的角色和该任务输出的工作产品拥有者之间存在一定的相关性。例如，**业务分析人员**负责**收集利益相关者需求**的任务，他们还是**利益相关者需求**这个工作产品的拥有者。

3.2.3 活动

一个**活动**代表一组任务。架构师执行如图 3.5 所示的活动中的任务，我们将在第 8 章和第 9 章中详细说明它们。

图 3.5 架构相关的活动

3.2.4 任务

任务是在项目的上下文中提供有意义结果的一个工作单元。它有明确的目的，通常涉及创建或更新工作产品。所有的任务都由适当的角色执行。

任务可能会被重复很多次，尤其当采用迭代开发方式的时候（在这一章后面将讨论迭代）。图 3.6 中列出了架构相关的任务和有关的主要角色，我们将在第 8 章和第 9 章中详细讨论这些任务。然而，您应该记住，这组任务仅代表了核心的架构设计任务，架构师通常还参与

其他任务，例如技术策略开发和技巧开发。正如您将看到的，架构师还给其他角色以帮助，例如检查开发流程，识别利益相关者，定义和排定需求优先级，评估及制订计划。

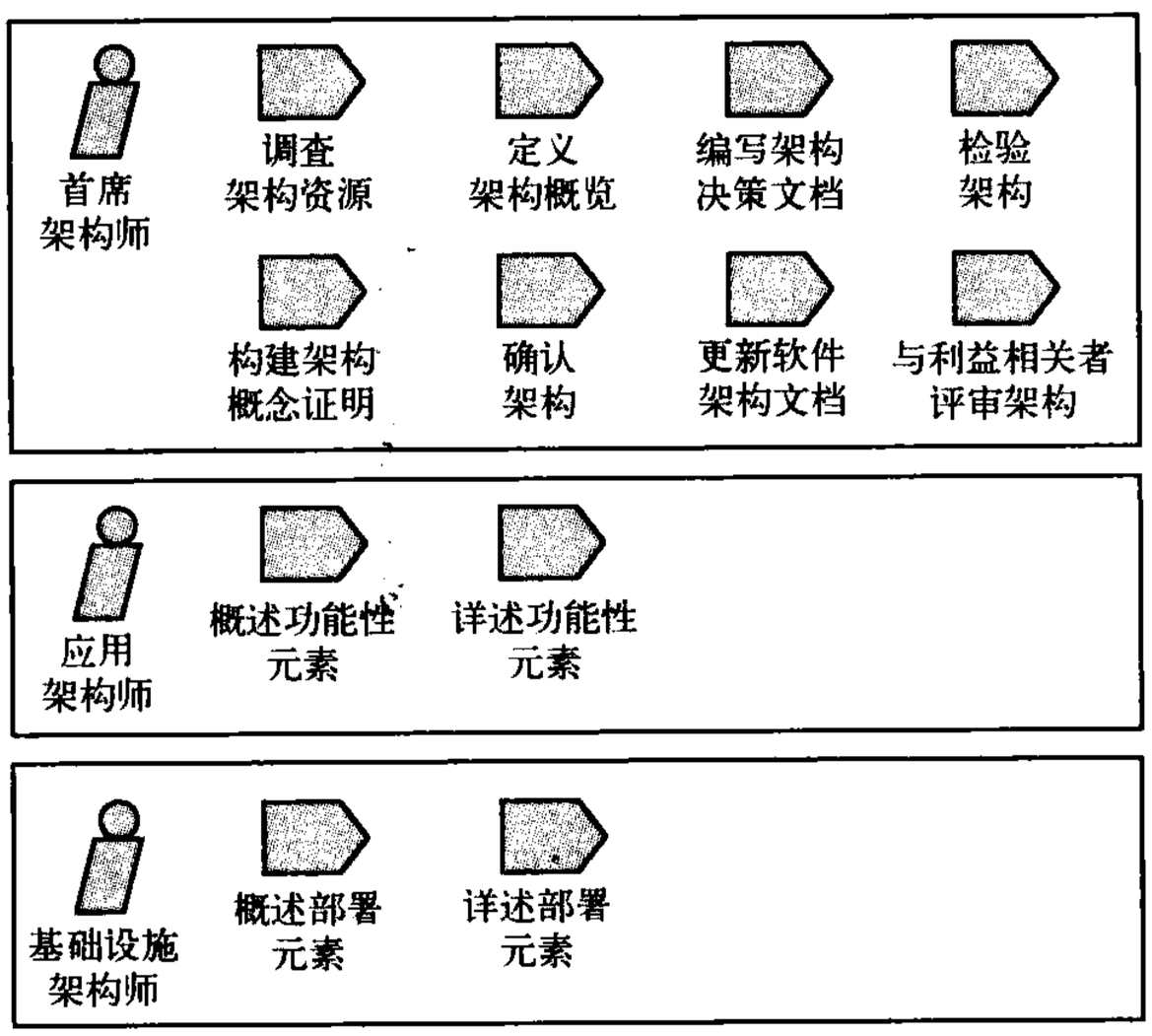


图 3.6 架构相关的任务

3.3 流程

现在我们把注意力转到方法内容的应用顺序上。正如您将看到的，软件行业中使用的各种方法之间的许多差异主要与遵循的流程有关，而不在于角色、工作产品、活动和执行的任务。

在这部分，我们将考虑三种类型的流程，它们分别具有瀑布、迭代和敏捷的特征。我们在讨论时不仅分析每种方法的关键特征，还突出最适合软件架构师的方法，探究一些似乎遍及这些讨论的迷惑。

3.3.1 瀑布流程

传统的瀑布开发流程如图 3.7 所示，其中使用了取自 OpenUP 的专业名称。在这种方式中，当一个阶段相应的工作产品被创建并被认可时，就认为这个阶段完成了。例如，当所有的需求都已经确认、详细定义且经过检查时，就认为需求阶段已经完成。然后，来自需求的输出会流入架构阶段。这个流程会一直持续，直到系统详细设计、编码（在开发阶段）、测试并对它的最终用户可用。工作产品中的修改（反向箭头表示）通常通过正式的修改流程处理。

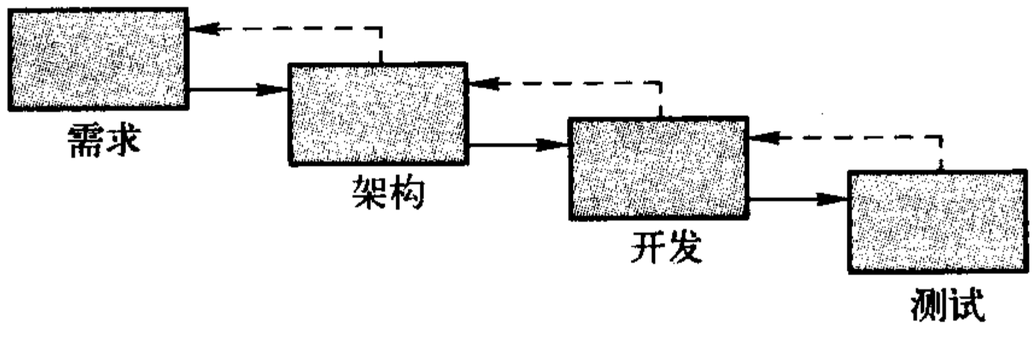


图 3.7 瀑布流程

这种方式被广泛使用，尤其在那些对一个现有系统进行较小增强的项目中，或在包含相对较少风险的系统开发中。然而，在绿色领域项目（其中架构师从头开始）或那些被大范围修改的项目上，这些方式的运用会有问题，原因如下：

- **项目进度不能精确地度量**，因为它是基于创建的工作产品而不是基于达成的结果。例如，暂时完成需求而没有做任何架构、开发或测试，就不能让您准确地估计项目将需要多长时间，因为您都不能真正地确定是否存在可行的解决方案！
- **直到项目的后期才能获得用户的反馈**，当系统可以使用的时候，延迟了实际需求的最终验收时间。
- **一些风险的解决方案延迟到项目后期**，在系统构建、集成和测试之后。这些活动通常会识别出设计中甚至规定需求中的缺陷，这就是遵循瀑布流程方式的项目容易产生进度延期的原因。

可以选择使用迭代开发流程，我们将在下一节讨论它。

3.3.2 迭代流程

> **迭代**是一个项目的短的周期划分。迭代使您能够证明增量价值并及早地获得持续反馈。（OpenUP 2008）

在一次迭代中，需要经历每个阶段，包括需求、架构、开发和测试。**迭代**是一个明确的、固定时间的活动序列，它们会产生一个可执行产品的（内部或外部）版本。随着项目的推进，这些版本会提供性能方面的逐步改善，直到最终的系统完成。迭代开发流程类似于栽培软件，在这个过程中，最终的产品随着时间逐渐成熟。每次迭代都导致对需求的更好理解、一个更强壮的架构、更有经验的开发组织和一个更完善的实现。通过一系列的迭代和可执行物来栽培一个架构的观念，似乎是我们认为在软件系统架构设计方面成功和高效的组织中的共同方法。

图 3.8 解释了项目的关注点如何移动到后续的迭代。在这个图中，您将看到在每次迭代中对每个专业都进行了处理，在每个专业内的方框大小表示在那个专业内执行活动（和任务）所花的相对时间。在这个简单的例子中，您可以看到，迭代 1 关注需求定义，除此之外还进行了一些架构设计（这次迭代中关注最高优先级的需求），同时还有一些开发和测试。在这个例子中，迭代 2 关注架构的稳定，这就是您看到重点在于架构设计活动的原因。迭代 3 关注基于相对稳定的需求和架构的基础上完善解决方案，这就是您看到重点在于开发和测试的原因。

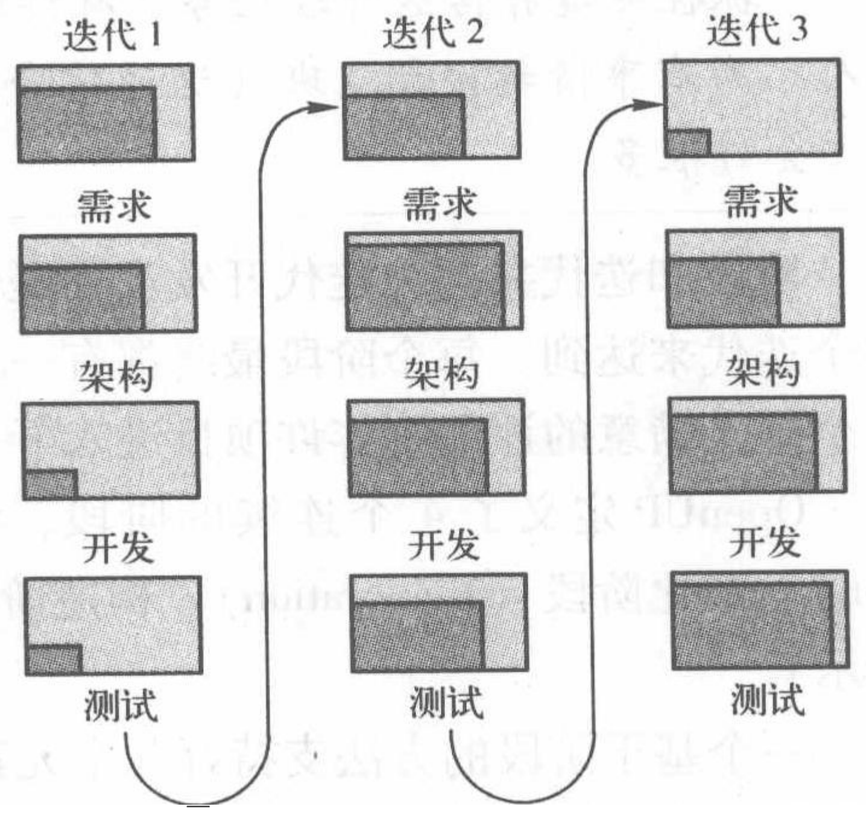


图 3.8 迭代流程

下面是遵循迭代流程取得成功的一些项目中的重要迭代特征：

- 迭代有明确的评估标准。

- 迭代要有一个计划的可论证性能。
- 迭代以一个较小的里程碑结束，在那里，迭代结果要相对于那个迭代的目标成功标准进行评定。
- 在迭代过程中，更新工作产品（工作产品同系统一起发展）。
- 在迭代过程中，对系统进行集成和测试。

迭代开发方式对架构师非常有吸引力，因为它明确承认架构的调整需要作为项目的进展。当然，架构调整的数量应该随着时间减少。要点是不能把这些调整看作是事后补救，而是项目生命周期的基本方面。

然而，一个迭代开发流程不仅包括一系列迭代，它还必须有一个所有迭代都在其中进行的整体框架，这代表项目的战略计划并推动每次迭代的目标。这样的一个框架按阶段（phase）提供。

> **阶段**是一个专门的活动类型，它代表项目中以一个决策点、主要里程碑或一组可交付物结束的一个重要时期。阶段通常拥有良好定义的目标并为如何组织项目工作提供基础。（OpenUP 2008）

阶段提供了定义良好的业务里程碑，它确保迭代向前推进并汇集在一个解决方案上而不是不确定地重复。迭代具有（固定周期的）固定时间，而阶段以目标为基础。阶段没有固定时间，因为一个阶段的完成是基于项目的状态评定的。正如后面的“缺陷：过快地宣告胜利”部分所讨论的，一些组织未能承认这个重要的原则。

缺陷：过快地宣告胜利

对于**项目经理**来说，非常普遍的一个错误是当一个阶段实际还没有完成时就宣告其已经完成，仅仅因为已经到了这个阶段应该完成的日期。这种情形类似于软件方面的假账，如此作假的**项目经理**通常变得很紧张，因为他们误导的决策会反过来困扰他们。

抓住关键并传达阶段任务没有按计划完成的消息，而不是迷惑他们自己（及其他所有人）所有事情都按部就班（这通常会给项目带来额外和不必要的压力），对于项目经理而言会好很多。

阶段和迭代共同为迭代开发流程提供基础。每个阶段的目标通过执行这个阶段内的一个或多个迭代来达到。每个阶段最终都有一个主要里程碑和确定这个阶段的目标是否达成的评估。一个令人满意的评估将容许项目进入下一阶段。

OpenUP 定义了 4 个连续的阶段，我们在本书中也使用这 4 个阶段：起始阶段（Inception）、细化阶段（Elaboration）、构造阶段（Construction）、移交阶段（Transition）（如图 3.9 所示）。

一个基于阶段的方法支持好几个元素随着项目的进行逐渐收敛。例如，风险随着项目的生命周期逐渐降低，成本和计划评估变得更精确。架构也随着时间变得稳定，如图 3.10 所示。碰巧的是，大部分基于阶段的软件开发流程都有一个关注于稳定架构的阶段。在本书中，这个

阶段就是细化阶段，很明显，它是架构师特别关注的阶段。

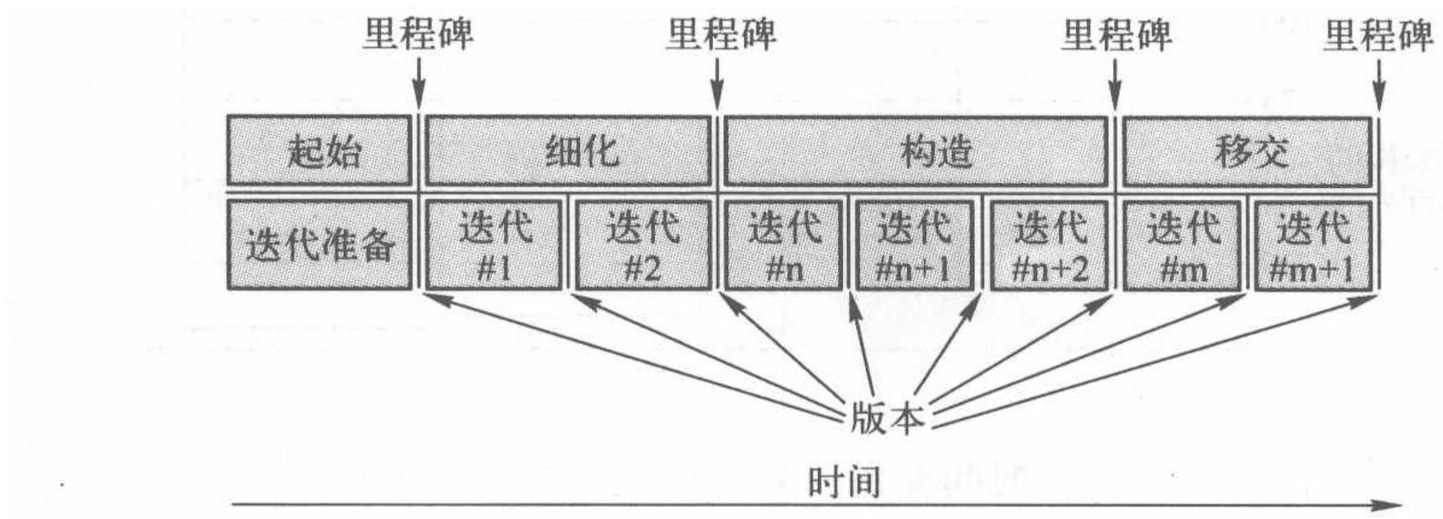


图 3.9　项目阶段

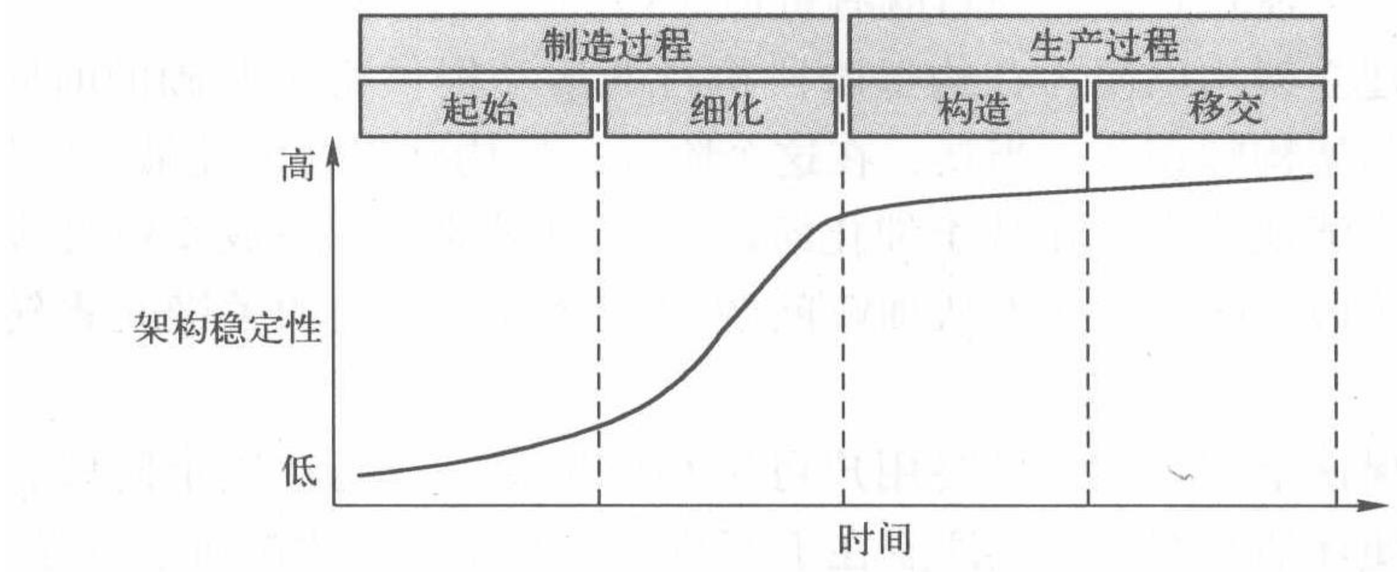


图 3.10　架构稳定性随着时间的变化

图 3.10 还把起始和细化阶段归类为制造过程，把构造和移交阶段归类为生产过程。其中的区别在《Software Project Management：A Unified Framework》（Royce 1998）中进行了详细讨论。从细化阶段移到构造阶段（就是从制造过程移到生产过程）特别有趣，因为这个移动代表了一个重要的里程碑：从此开始关注业务。生产过程通常比制造过程更可预测，因为通常对问题和解决方案都有很好的理解。因此，在成本和进度评估方面有高得多的真实度，也可以更自信地进行分配资金决策。

这并不是说到细化阶段末所有需求都已经定义好或需求再也不作改变，而仅仅认为主要的需求已经定义好，任何改变对架构都只有细微的影响。当出现异常时（例如，增加一个重要的需求），可以检查这个项目及它的预定时间表并作相应的调整。这可能需要从根本上重新安排这个项目并再次进入细化阶段（甚至起始阶段）。

如图 3.11 所示，在每个阶段架构的重点和用于架构的资源数量（通常会根据所开发系统的种类不同而变化）之间存在一个直接的关系。这个图是根据《Software Project Management：A Unified Framework》（Royce 1998）中的信息得出来的。

很明显，从一个阶段移到另一个阶段是项目生命周期中的一个重要时刻，因此，理解决定从一个阶段移到另一个阶段的标准很重要。附录 C 将提供每个阶段的主要目标和它们的里程碑评估标准的概要信息。下面是每个阶段的概述：

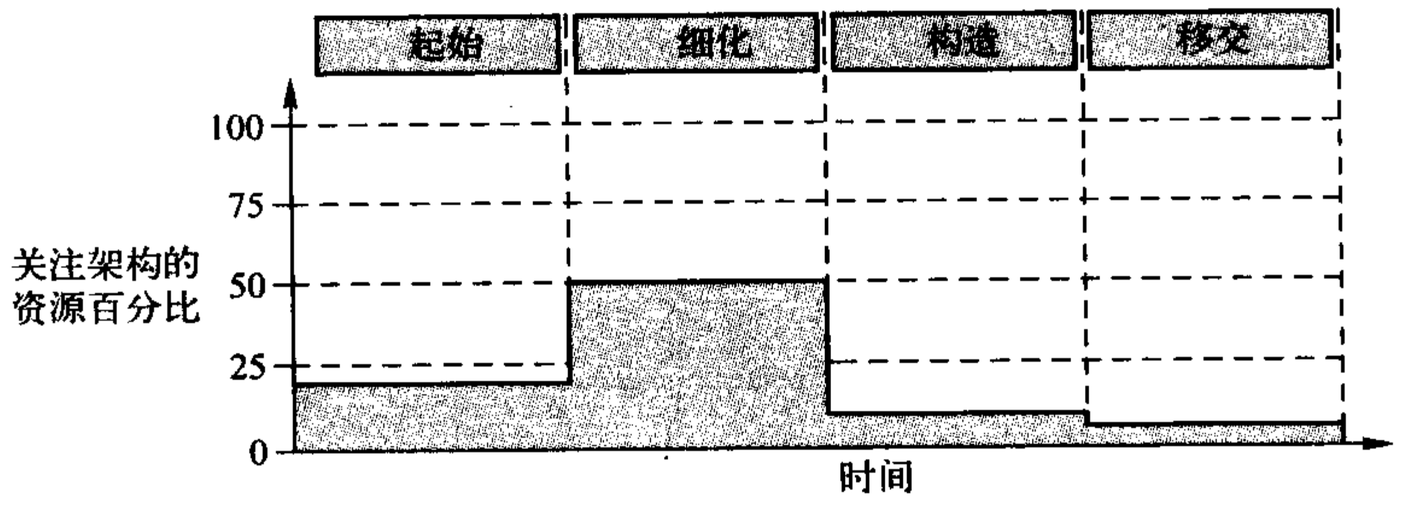


图 3.11 随着时间变化的关注架构的资源百分比

- **起始阶段**是确定项目的业务案例及确定利益相关者对项目的目标达成一致的时间段。在起始阶段，关注点是确保项目既有价值又切实可行。
- **细化阶段**是建立架构以便为在构造阶段执行的活动提供稳定基础的时间段。因此，这个阶段尤其与架构师相关，当然，在这个阶段，架构师需要消耗最多的精力。
- **构造阶段**是澄清遗留需求和基于细化阶段确定的基础架构完成系统开发的时间段。在细化和构造阶段之间，关注点从理解问题和确定解决方案的关键元素转移到开发一个可部署产品。
- **移交阶段**确保这个软件对于最终用户可用和可接受。就是在这个阶段，系统被部署到用户环境以便评估和测试，关注点在于调整这个产品和解决配置、安装和可用性问题。到移交阶段末期，这个项目应该可以结束了。

迭代开发和瀑布开发

与瀑布开发流程相比，迭代开发流程是以结果驱动的（它关注特定迭代中达到的结果），而不是以工作产品驱动的，这有助于对项目进度进行更精确的测量。迭代的方式通过系统增量版本的执行尽早获得用户反馈，而这些反馈促进实际需求的收敛。另外，迭代方式还确保在每次迭代内进行集成和测试，确保生成一个可执行的版本。在项目早期就会遭遇风险，因此，在对项目作出充分的投资决策之前，支持确定中的业务。正如 Royce 提出的：

> 尽早建立一个抵御风险的迭代生命周期流程。对于当今非常复杂的软件系统，按次序地定义整个问题，设计完整的解决方案，构建软件，然后测试最终产品，这是不可能的。相反地，一个对问题的理解、有效解决方案和包含好几次迭代的有效计划进行精炼的迭代流程有助于均衡对待所有利益相关者的目标。主要的风险必须尽早处理以增加可预测性和避免昂贵的下游紊乱和重写。（Royce 1998）

3.3.3 敏捷流程

最近几年，人们对于敏捷流程的兴趣逐渐增加，代表性的方法有极限编程、Scrum、精益和特征驱动开发。虽然每个特定的敏捷方法以及其倡导的方法存在一些差异，但是它们都基于相同的基础原则。这些原则的一个体现是《敏捷宣言》（《Agile Manifesto》，2009），它申明了下列价值观：

- 注重个人及其交互，胜于过程与工具。
- 注重可用的软件，胜于详尽的文档。
- 注重客户协作，胜于契约谈判。
- 注重响应变化，胜于恪守计划。

作者们的观点是：这些原则是迭代开发方法中发现的原则的补充。另外，正如您可以从《敏捷宣言》中看到的，主要的区别是原则的重点在于指导流程的执行，而不是新的或不同的方法内容。Scrum 是敏捷流程的一个例子：

> Scrum 是关注于构建软件达到业务需求且避开复杂性的一个管理和控制流程。Scrum 封装且超越了现有工程实践、开发方法学和标准。(Schwaber 2002)

Scrum 还举例说明了流程的重点。它关注迭代的内容（把迭代看作是 Sprint）、迭代中（Sprint Backlog）考虑的排定优先级的需求和简短的每日状况会议（Scrum）。

敏捷原则还与架构师有关，令人惊讶的是这些原则经常被误传。一个共识是敏捷流程不倡导预先设计，架构直接来自于代码。然而，“注重可用的软件，胜于详尽的文档”这个原则并不意味着没有文档（包括架构相关的文档），这仅仅意味着只有满足当前迭代目标的文档。

> 在敏捷的团体中似乎存在这样一种担心：如果我们使用“模型”或“文档”这样的术语，“有害的官僚主义”就会干扰我们的项目，迫使我们编写很多详细的需求规范或迫使我们采取预先设计的方式……奇怪的是，实际上，敏捷开发人员也定期建模，即使他们没有直接谈论它。(Ambler 2008)

3.4 总结

这一章概要介绍了本书中使用的方法基本原理，利用软件系统流程工程元模型规范（SPEM）标准作为一个框架来解释这些概念。这一章强调在方法内容和实现这些内容的流程之间的区别，还讨论了不同类型的流程，包括瀑布、迭代和敏捷。这一章讨论了这些概念与架构师的关系和与架构设计流程的关系。本书中使用的方法都来自这些类型的流程。

下面两章（第 4 章和第 5 章）关注值得进行更详细讨论和在项目生命周期中常常使用的方法的特定方面。

第 4 章

编写软件架构文档

编写软件架构文档的主要目的是使架构能进行沟通。在确保所有的利益相关者理解架构并相应地提供他们的输入信息方面，这种沟通很有必要。这种沟通对于确保某些利益相关者对提出的解决方案感到满意（例如，项目组对所构建的系统有一致观点）很关键。可是，适应所有利益相关者的所有关注通常是具有挑战性的。因此，我们在这一章中探索描述一个软件架构的不同方面。编写软件架构文档有如下好处：

- 有文档的架构有助于不同的利益相关者之间进行有效的沟通，如那些开发系统的利益相关者和那些维护系统的利益相关者之间的沟通，甚至是那些相关的但未曾谋面的人之间的沟通。
- 有文档的架构可以提供追溯其他工作产品的上下文，如详细的设计工作产品乃至培训材料。
- 有文档的架构可以传达可供选择的架构解决方案，并提供每个方案赞成和反对的理由。
- 有文档的架构有助于从一个现有架构转换到一个新架构的计划编制。
- 有文档的架构通常能通过识别组成架构的元素及它们之间的依赖性来帮助编制计划。这些信息有助于制定一个恰当的实现这种架构的顺序，并且也有助于理解对架构进行调整所带来的影响。
- 有文档的架构可以提醒架构师在其所作的某些决定背后的基本原理。
- 在可重用资源方面，有文档的架构有助于识别哪些资源可重用以及可重用的时机。
- 有文档的架构有助于架构的评估，例如，它有助于确定所做的实现是否与规定的需求说明书相一致。

编写一个软件架构文档时所用的一些简单的概念对于编写架构文档经验很少的架构师特别有用。在这一章中，我们先讨论这些概念，然后讨论几个架构描述框架（ADF），最后讨论本书中使用的架构描述框架。

4.1 最终的结局

在开始讨论那些组成文档的软件架构的元素之前，我们先介绍一下我们将使用的一个非常简单的例子，如图 4.1 所示。

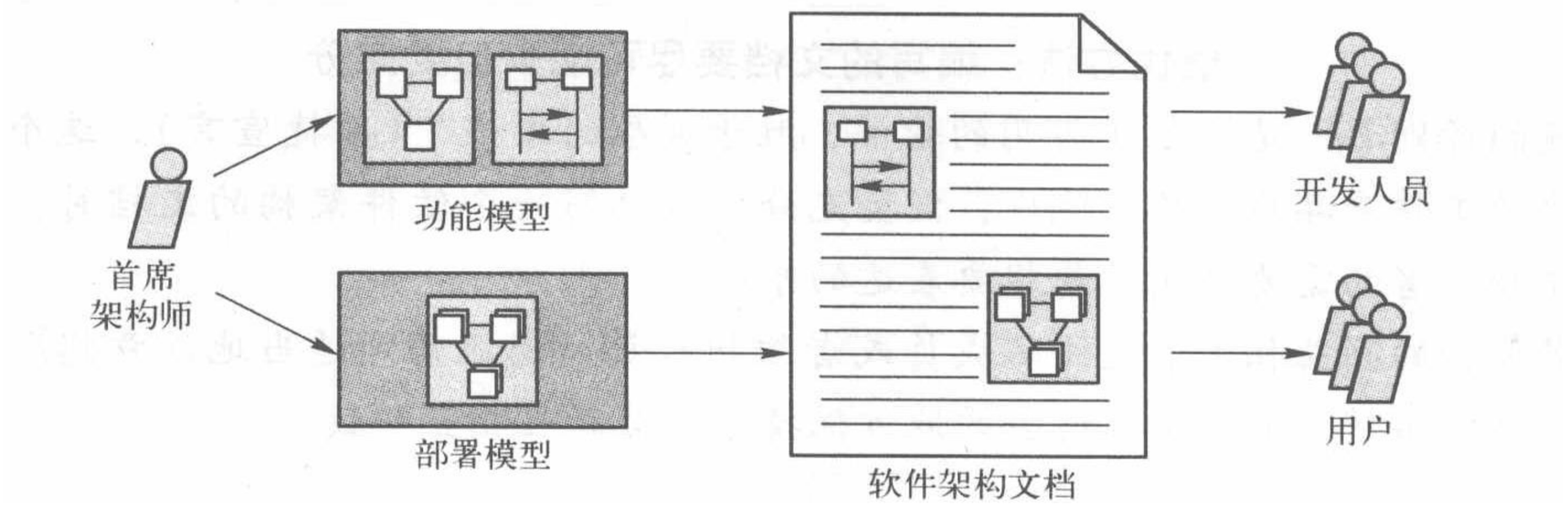


图 4.1　与编写软件架构文档有关的元素

当编写一个软件架构文档时，您应当遵循一些简单的步骤：

- **识别利益相关者组**。在编写软件架构文档中，首先了解的是目标对象。首席架构师首先考虑的事情之一是用架构利益相关者表示的目标对象。利益相关者可以按具有共同关注点的团体来进行组织。在图 4.1 中，开发人员和用户就是两个利益相关者组。
- **选择视点**。在识别利益相关者组之后，您必须识别那些能够使利益相关者的关注得以表达的相应视点。视点对于编写软件架构文档很重要，在 4.3 节中将对它进行详细讨论。例如，一个视点详细定义了必需的工作产品以便能够与识别的利益相关者用适当的方式对架构进行交流。在图 4.1 的例子中，功能性视点选项要求架构师创建一个**功能模型**来描述系统在某些方面（如组件、组件之间的关系和它们的交互）的功能性（逻辑）。部署视点选项要求架构师创建一个**部署模型**来描述系统在某些方面（如节点和它们之间的连接）的部署。为简单起见，图 4.1 中没有显示其他视点和它们相应的工作产品（如架构决策和**架构概念验证**工作产品）。
- **创建工作产品**。架构师（或者架构团队）相应地组装工作产品。您可以给您决定创建的任何模式增加元素，也可以增加任何容许您可视地表达这些元素的图表。在图 4.1 中，我们在**功能模型**中放置了两个图表，在**部署模型**中放置了一个图表。总而言之，任何工作产品都可以看成是这个开发中系统的一个模型；我们将在下一节中讨论这个主题。
- **给架构描述打包**。架构师通常不是利用已经创建的所有工作产品来沟通架构，而是特别地采用一种更容易由利益相关者使用的方式（例如，不是所有的利益相关者都有权使用用于创建工作产品的所有建模工具）打包元素。在图 4.1 所示的例子中，架构师创建了一个可交付使用的**软件架构文档**。这个软件架构文档是按架构的不同视图组织起来的，每个视图代表一个特定系统的一个特定视点的应用。架构文档的必需编写范围在后面的“最优方法：编写的文档要尽可能少但要充分”中讨论。

视点（viewpoint）、**视图（view）**和**模型**的概念将在下一节详细讨论。

最优方法：编写的文档要尽可能少但要充分

敏捷的原则之一是“注重可用的软件，胜于详尽的文档”（敏捷宣言），这个原则提醒您：创建的工作产品应该尽可能少，但要充分。当编写一个软件架构的文档时，通过充分考虑利益相关者的需要来决定您想要表达的东西。

倘若架构的利益相关者是团队成员或者维护人员，您还必须适当地注意您所创建的任何工作软件的维护，要确保在适当的地方保持它们最新且相互一致。

4.2 关键概念

架构描述的关键概念是**视点**、**视图**和**模型**。这些概念在 IEEE（电气和电子工程师协会）1471 2000（电气和电子工程师协会针对软件密集系统架构描述制定的操作规程建议）中进行了定义。图 4.2 中列出了这个标准中定义的部分元素，这个图通过明确确定这些与一个架构描述相关的元素来进一步阐述第 2 章“架构、架构师、架构设计”中的讨论。

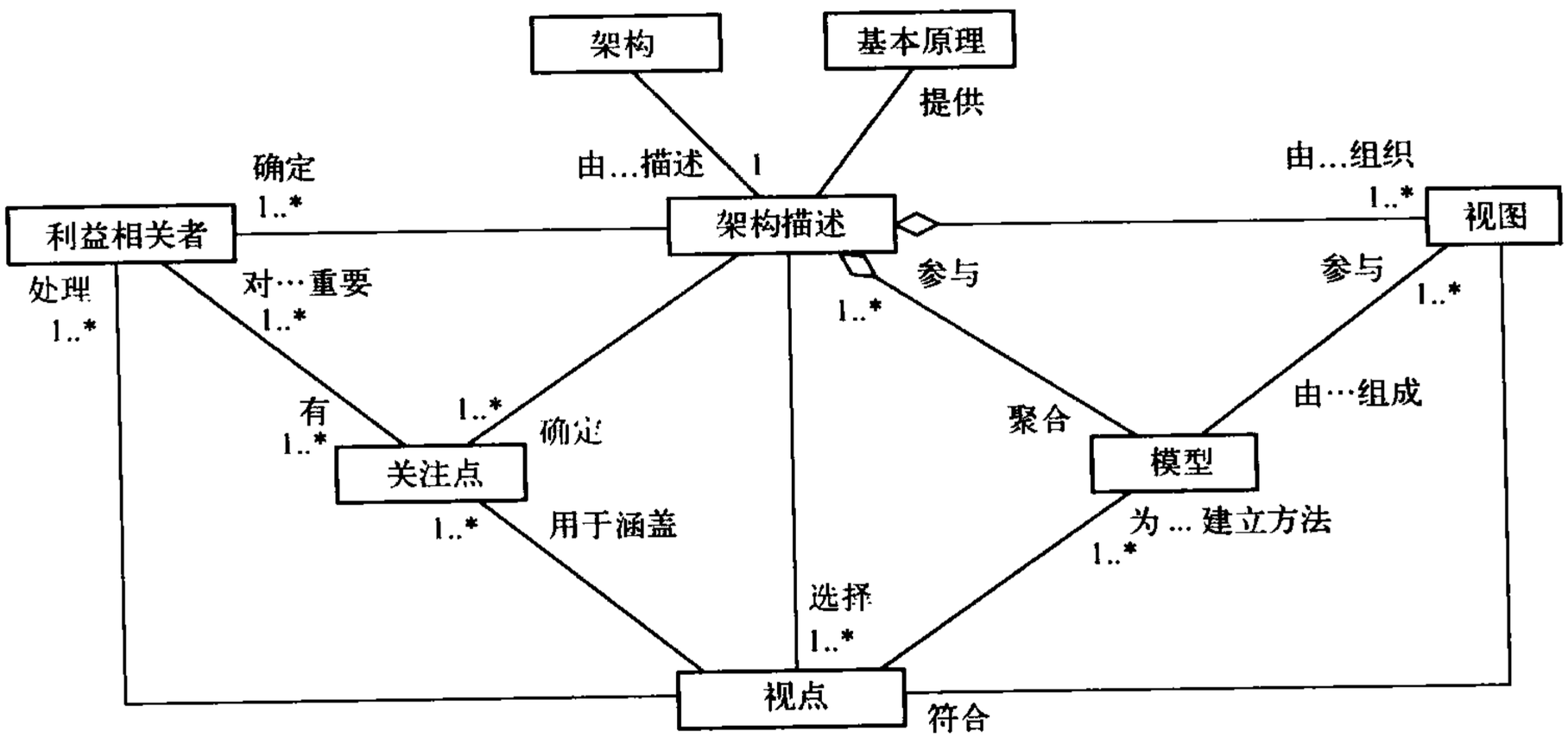


图 4.2 与架构描述相关的元素的元模型

图 4.2 中显示的所有关系都来自电气和电子工程师协会（IEEE）1471 标准。第 2 章中未进行描述的那些关系都列在了下面，我们稍后在这一章中将详细阐述这些关系。

- 一个架构描述是通过一个或多个视图来组织的。
- 一个视图由一个或多个模型组成。
- 一个模型参与一个或多个视图。
- 一个模型参与一个或多个架构描述。
- 一个视图符合一个视点。
- 一个架构描述选择一个或多个视点。
- 一个视点为一个或多个模型建立方法。

- 一个视点用于涵盖一个或多个关注点。

正如您所见，**视点**、**视图**和**模型**是描述一个软件架构的基础，在随后的部分将对它们进行更详细地描述。

4.3 视点和视图

大家都非常希望能够通过单个满足所有利益相关者需要的图就能沟通一个系统的架构。虽然这可能是与利益相关团体非正式地沟通架构的某些方面的一种有效方法，但是，采用这样有限的一个方法（一个想要包含架构的所有方面的非正式技术说明）通常会引起混乱，因为这种图一般都过度复杂。这类图也会混合架构的不同方面。例如，在这样的一个图中，一个看似简单的方框可能混合了功能职责和部署关注点，而在方框之间的箭头可能混合了控制流和元素之间简单的依赖。另一个常见的缺点是这种方法强调了架构的一个方面而损害了其他方面。例如，这种图可能强调了把架构功能划分为组件、它们的关系和交互作用，而损害了关于系统部署的关注点。

这些缺陷的解决方案是同时应用几个视点来描述一个软件架构，每个视点处理一组特定的关注点。

> 架构的**视点**是用于构建和利用一个视图的常规说明书。它是一个模式或模板，可以利用它确定一个视图的目标和用户及其创建和分析的技术，从而来开发单独的视图。（IEEE 1471 2000）
>
> 架构的视图是从一组相关关注点的视角看整个系统的一种表示法。（IEEE 1471 2000）

视点有几个值得注意的特点：

- **一个视点有一个明确的观众**。一个视点的主要关注点是它处理一个特定观众的需要，在这里，观众所指的是一个或多个利益相关者。一个功能性视点可以处理开发人员（他必须承担详细的设计活动和实现代码）的需要、测试人员（他必须对系统进行测试）的需要等。
- 一个**视点是为了达到一个或多个目的**。虽然一个视点支持一组拥有类似关注点的利益相关者，但是，那些关注点可能是不一样的。一个开发人员可能会在用特定语言实现某一特定设计元素之前利用一个功能性视点来理解设计该元素内部的上下文。然而，一个测试人员可能使用相同的功能性视点来理解这个系统的关键组件，以便确定这个系统是否满足某些需求。
- **视点是视图的一个模式或模板**。简而言之，一个视图符合一个视点。视点定义了视图创建和使用的规则。用面向对象的术语来说，视点代表的是一个（视图的）类，而视图代表的是视点的一个特定实例。因此，当编写一个特定系统的架构文档时，您选择一组适当的视点，最终会创建专门针对这个开发中系统的架构的视图。因为可以把视点看作一个模板，所以，一些组织会投资创建任何项目都可以从中选择的**视点目录**一

点也不奇怪。

- **视点定义技术**。最后，视点定义了创建、描述和分析一个视图的规定。视点定义了用来定义符合这个视点的视图的语言，也许还包括用于追溯这个视图内容的注释和特定的技术。一个部署视点可能会规定一个部署视图包含一个**部署模型**；规定采用统一建模语言（UML）创建这个**部署模型**；规定采用特定的技术和方法来创建这个**部署模型**；还规定这个**部署模型**应该应用于某些情形（会获得相应的好处并避免缺陷）。由此可见，视点与组成这个视图的工作产品的创建方式有关。

4.3.1 基础视点

我们已经提及了架构师在实践中经常遇到的两种视点：一种是功能性视点，它关注支持系统功能性的元素（如组件、它们的关系和它们的行为）；另一种是部署视点，它关注支持系统分布的元素（如节点、设备和它们之间的联系）。

考虑视点的基本原理可能进一步延伸，如图4.3所示。在图中，位于架构描述边缘的视点是：需求视点和确认视点。需求视点的目的是为形成这种架构的系统需求提供一个说明，它包括功能性需求、品质和约束。确认视点的目的是为系统是否提供了必需的功能、展示必需的品质和适应定义的约束提供一个说明。

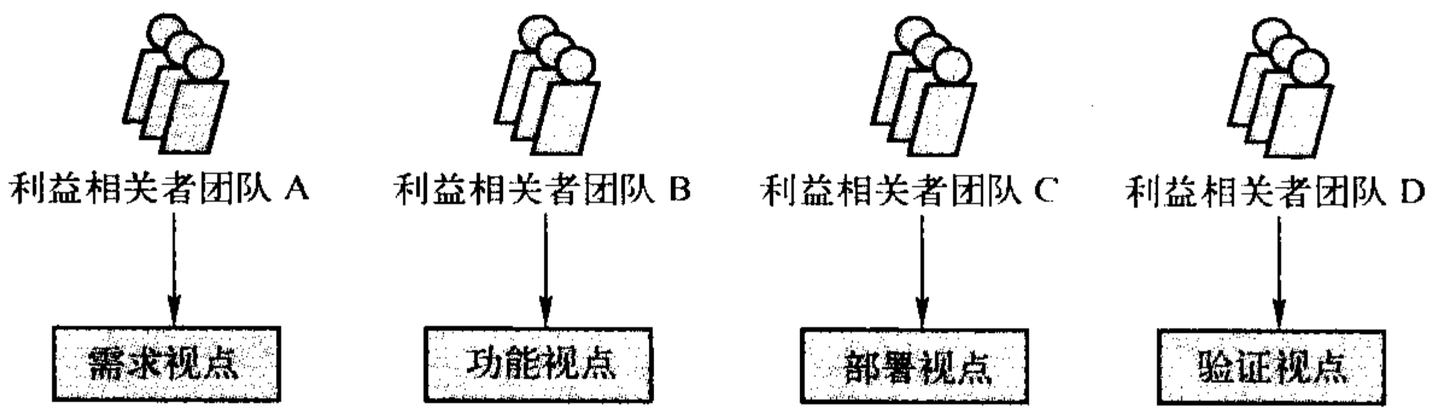


图4.3 基础视点

4.3.2 交叉视点

在Nick Rozanski和Eoin Woods的《Software Systems Architecture：Working with Stakeholders Vsing Viewpoints and Perspectives》（Rozanski 2005）本书中，他们给**视角（perspective）**这个术语一个专门的含义，用来代表那些处理交叉关注的元素集。

> 架构**视角**是用来确保一个系统展示出一组特定的需要从许多系统架构视图考虑的品质相关属性的一些活动、策略和指导方针的集合。（Rozanski 2005）

视角实际上是关注系统必须具有的品质的一类特定的视点。例如，为了处理一个系统中的安全性需求，您必须在解决方案中详细说明安全性需求，引入与安全性相关的功能性元素和部署元素，随后验证架构中实现的安全性。这个情形如图4.4所示。在这个图中，我们介绍了一个性能视点和一个安全性视点。通过应用这些视点而创建的相应的视图被认为是跨越了需求、功能、部署和验证的视图，在它们之间可能共享了相同的元素。更明确地说，对于图4.4所示

的例子，您可以考虑由这些视点形成的下列交叉点：

- 性能视图确定需求视图中与性能相关的需求和这些需求实现的方式。
- 性能视图确定功能视图中有助于满足性能需求的元素，例如，当大量通信发生时，这将可能导致把元素紧紧地耦合在一起。
- 性能视图确定部署视图中有助于满足性能需求的元素。这个视图会考虑通信元素的位置、支持系统的硬件规格和从总体上考虑系统的分布（例如，考虑像网络延迟这样的因素）。
- 性能视图确定验证视图中支持性能需求验证的元素。这个视图会考虑架构对必要的性能特性的满足程度、进行折中的基本原则和突出的风险和问题。
- 安全性视图确定需求视图中与安全相关的需求和这些需求的实现方式。安全性需求实际上可能是功能性的，例如用户如何让系统验证自己，也可能是对系统所设置的限制，例如用于对系统中的数据进行加密的加密强度。
- 安全性视图确定功能性视图中有助于满足安全性需求的元素，例如处理认证、授权、审核和认可的组件。这个视图可能还会列出必须保护的组件和数据元素。
- 安全性视图确定部署视图中有助于满足安全性需求的元素。这个视图可能会确定需要专门的硬件或者软件（如防火墙）。
- 安全性视图确定验证视图中支持安全性需求验证的元素。这个视图会考虑架构对必要的安全特性的满足程度、进行折中的基本原则和突出的风险和问题。

	需求视点	功能性视点	部署视点	验证视点
性能视点	性能需求	组件耦合	组件位置 硬件规格 分布拓扑	性能验证元素
安全性视点	安全性需求	安全性策略 用户认证 用户授权	防火墙	安全性验证元素

图 4.4　基本视点和交叉视点

除考虑质量外，交叉视点的考虑还能进一步延伸。您可能决定引入一个基础设施视点，它关注那些对开发中的系统提供支持但与业务领域无关的元素（如错误日志或者持久机制）。相应的视图又会跨越基础视图。您可能有与基础设施相关的需求（如使用特定关系型数据库的约束）、与基础设施相关的功能性元素（如一个错误记录的机制）、与基础设施相关的部署元素（如用于承载这个错误记录机制的硬件节点）和与基础设施相关的验证元素（如这个错误记录

机制对规定需求处理的完善程度的评估）。

实际上，作者们已经发现同时关注基础视图和交叉视图可以为我们提供一个简单的模型来考虑架构的某些方面，还可以确保在项目的整个生命周期中能把适当的重点放在关键的架构特性上。

4.3.3 视图及图表

架构师中一个常见的误解是认为视图（view）就是图表（diagram）或者图表就是视图，造成这种误解的主要原因是IT领域中这两个术语的相似性。虽然一个视图可能通过一个简单的图表就能完整地进行描绘，但是，更常见的是一个视图需要许多图表来进行描绘。

考虑一架波音747巨型喷气式飞机，它由重达几十万磅的超过6百万个部件组成，包含160多米的配线，还有一个含有好几百个控制项目的驾驶员座舱，它可以直接把好几百名乘客运达几千公里之外。在19世纪60年代设计第一架波音747飞机时，波音公司绘制了75000多张工程图纸。如果每个图表都是一个视图，那么，结果不仅是视图的激增（因为会涉及很多图表），而且也会产生多得难以处理的图表。因此，不同的图表经常瞄准相同的关注点，应该被归为一组。根据我们的定义，每组图表都可以看作是一个视图。其实，一个视图是从一个特别有利的位置观察整个架构的一个窗口。

视图和图表之间的关系在图4.5中进行了简要的说明，在该图中列出了两个视图：一个功能性视图和一个部署视图。在这个例子中，与利益相关者团队A的关注点有关的这个功能性视图关注组件和组件之间的交互，它由两张图表组成。与利益相关者团队B的关注点有关的这个部署视图关注节点和节点之间连接方面的部署拓扑，它由单个图表组成。

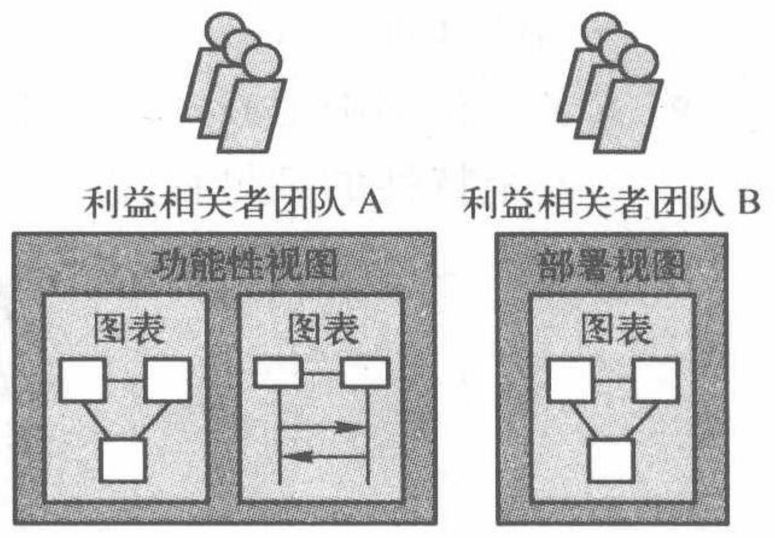


图4.5 一个视图和图表的例子

4.3.4 视点及视图的优点

除了编写软件架构文档常见的好处之外，采取基于视点和视图的方法可以获得一些特别的好处：

- **可以管理系统的复杂性**。一个视图代表的是实体的一个简化。这种简化是通过关注考虑中系统的特定方面而获得的。飞机的物理视图可能关注它的空气动力学的特性，而一个软件系统的部署视图关注系统将在其中运行的操作环境。
- **可以关注系统的特定方面**。大多数复杂的系统都有好几个相关联但又有所不同的视图。一架飞机很明显有一些外部结构，如机翼、机身和起落架，电路，水力系统等。同样地，一个软件系统也有功能性特性、部署特性等。视图很清晰地分离了这些不同的关注点。这种分离使您能够考虑整个系统的局部，而不会因为整个系统的复杂性而感到困扰。

- **可以和利益相关者进行交流**。为了能够与一个系统的不同利益相关者进行有效地交流，您常常不得不用不同的视图来描述同一个系统的不同方面以处理每个利益相关者的需要。考虑飞机外部结构的一个视图可能用于和空气动力学工程师进行交流。同样地，软件系统的一个部署视图可能用于和设计人员及系统管理人员进行交流。

4.4 模型

模型是在 IEEE 1471 标准中用来表示参与一个视图的工作产品的术语。模型中会包含显示在一个视图中的特定元素。飞机模型就是一个很好的例子：为了更好地了解飞机的空气动力学特性，会把这个模型放置在一个风洞里面。正如您将看到的，就目的而言，这个例子和用于表示一个软件系统不同方面的模型之间没有区别，因为模型代表的是正在考虑中的系统的一个抽象。

模型提供了一个架构的特定描述或内容。例如，一个结构视图会由一组系统结构的模型组成。这种模型的元素可以包括可识别的系统组件、它们的接口以及那些组件之间的联系。

本书中描述的模型包括**功能性模型**、**部署模型**和**数据模型**。另外，在一般意义上，甚至在文档或者电子数据表中描绘的工作产品也可以看作是模型，因为它们符合前面章节中给出的模型定义。

图 4.6 显示了视图、图表和模型之间的关系。正如您可以从这张表中看到的，模型可以被多个视图共享。确切地说，虽然功能性视图显示**功能性模型**中的元素，而部署视图显示**部署模型**中的元素，但是，您也可以看到，部署视图也显示了**功能性模型**中的元素，因为我们要把组件的部署（来自**功能性模型**）表示到节点（来自**部署模型**）上。

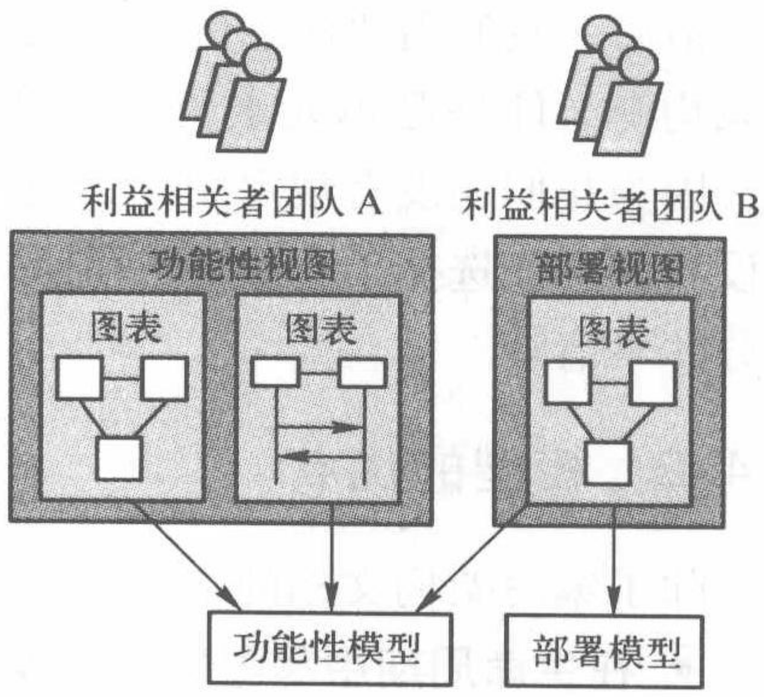


图 4.6 视图、图表和模型的例子

4.4.1 实现的层级

虽然不同的模型用于记录系统的不同方面，但是，一个特定模型自身都可能需要经历一系列的改进，尤其是描述解决方案方面的模型，如**功能性模型**这个工作产品。一个模型在它的生命周期中所经历的不同的实现程度可以使其逐渐从一个初始模型（其内容是非常概念性的）演变成一个足以用来作为实现的基础的模型。

举例来说，既从逻辑（或与技术无关的）方面又从物理（或特定技术）方面来考虑**功能性模型**这个工作产品，这是有可能的。同一个模型的这两个方面描述的是这个模型不同的实现层级。

这样的思想也可以应用到其他模型上。例如，可以在逻辑数据模型（识别实体和它们之间

关系的模型）方面或者在物理数据模型（这个模型还确定实现的关注点，如改善数据存取性能的索引表和存储过程）方面都进行这样的考虑。这个思想反映在图 4.7 中。如果适合的话，架构师还可能选择保留逻辑模型并创建独立的物理模型。第 8 章中会进一步讨论这个主题。

要注意的是，实现的层级和抽象的层级是不一样的。抽象的层级与细节的数量有关，抽象的层级越高，那么牵涉的细节就越少。然而，当考虑实现的层级时，您不能简单地增加或者删除细节，而要从根本上做出一些决策，从而导致考虑不同的元素。

视图 / 层级	功能性视图	部署视图
逻辑架构	功能性模型 数据模型	部署模型
物理架构	功能性模型 数据模型	部署模型

图 4.7 视图、模型和实现的层级

例如，在逻辑功能性模型中表示一个订单管理器的组件在 Java EE 环境中可能会作为一个 Enterprise JavaBean（EJB）来实现，并在物理功能性模型中提供 Java EE 特定的接口。因此，当从逻辑架构迁移到物理架构时，架构师必须向那些对考虑使用的技术很熟悉的技术专家请教，因为取决于使用的技术（和模式），在逻辑元素和物理元素之间可能存在一对一、一对多或者多对一的映射。这个主题在第 8 章中会进一步讨论。

最后，我们需要注意的是，在逻辑元素和物理元素之间的区别并不是黑白分明的。例如，负载均衡组件是逻辑元素吗？回答当然是“看情况而定”。这个组件当然不是针对特定技术的，因为我们还没有规定用什么技术来提供负载均衡。然而，它也不是与技术无关的，因为它假设我们已经选择了一个要求负载均衡的环境。总的来说，请记住，**逻辑**和**物理**这两个术语的区分并不清晰。

4.4.2 模型的优点

除了编写架构文档的常规好处之外，创建模型还有一些特定的好处：

- **在生命周期中尽早发现错误和疏忽**。要了解飞机的空气动力学性能，创建一个飞机的等比例模型并把它放到一个风洞中与建造一个真正的飞机并让它飞起来相比，前者要便宜很多。同样地，创建一个软件系统的设计模型并测试这个模型与构建真正的软件系统并来验证相比，前者要便宜很多。
- **检查不同选择的优缺点**。创建一个飞机的几个模型（都满足同样的要求），这样做使得每个解决方案的优缺点不必通过制造几个实际大小的飞机才能得到检验。同样地，从适当的模型中取得的一个软件系统的几个设计可以用来讨论提议的软件解决方案的优缺点。
- **了解变化的影响**。如果一个模型含有来自其他模型的元素（如来自需求的解决方案的元素），那么，为这些位于不同模型中的元素之间的关系编写文档是有可能的。这些关系通常称为**可追溯关系**，因为它们显示了一个模型中的元素如何追溯到另一个模型中的元素。然后，架构师可以利用这些信息来了解进行变化所产生的影响。例如，可以评估一个需求改变对于解决方案元素的影响。相反地，改变一个解决方案元素（如重写一个算法）

的影响也能通过查看这个解决方案元素全部或部分满足的需求范围来确定。

- **帮助编制项目计划**。在进度和资源方面，一个模型的元素可以帮助编制项目计划。拥有一个显示元素之间相互依赖关系的解决方案模型可以指明这些元素的实现次序、必需的技能和所需的精力。

4.5 架构描述框架的特征

简单地说，一个架构描述框架代表一组能用来共同描述一个架构的视点（和视图、模型、其他的工作产品、技术等）。架构师通常从一个视点目录选择视点并在必要的情况下调整它们以达到他或她的需要，而不是从头开始。视点的选择通常由一些组织的标准（例如，必须要有一个需求视点和一个功能性视点）以及考虑中系统的性质（例如，在单一节点上运行的系统就不会使用部署视点）来确定。

这一节通过调查现今使用中的某些架构描述框架中例证的特性来为在本书中使用的架构描述框架做好准备，也为先前的讨论增添一些内容。这一节并不打算对您可能遇到的所有架构描述框架进行全面的讨论——而仅仅讨论那些有助于证明一个特定特性的架构描述框架。这一节中要强调的特性是：

- **多重视图和场景视图的使用**，来自 Philippe Kruchten 的“软件架构的 4 + 1 视图模型”（Kruchten 1995）。
- **实现层级**，来自 Zachman Framework，由 John Zachman 创建（Zachman 1987）。
- **交叉关注**，来自《Software Systems Architecture：Working with Stakeholders Using Viewpoints and Perspectives》（Rozanski 2005）。

正如您将看到的，我们讨论的每个架构描述框架都关注于确保架构的许多方面都明确分离但又具有定义明确的关系，从而保持一致。好几个架构描述框架还包括过程指导的元素。在《Documenting Software Architectures：Views and Beyond》（Clements 2003）中还提供了如何有效地为一个软件架构编写文档的考虑。

在接下来的讨论中，记住对于任何给定的项目在视图和视点之间存在一一对应的关系（即使您把一个视图看作是一个视点的实例），这很重要，虽然不同的架构描述框架在它们的描述中宁愿使用一个术语。

4.5.1 软件架构的 4 +1 视图模型

一个普遍使用的架构描述框架是由 Philippe Kruchten 定义的“4 +1 软件架构视图模型”（Kruchten 1995）。这种架构描述框架定义了 5 个视图，如图 4.8 所示。

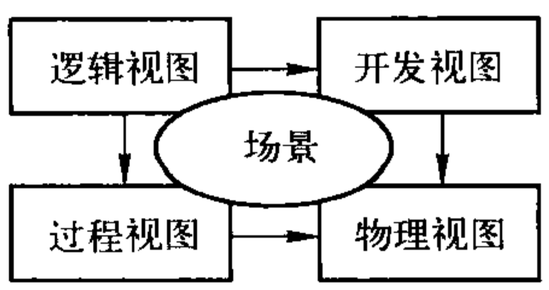


图 4.8 4 +1 软件架构视图模型

作者们对各种视点的描述如下所示：

- 逻辑视图是设计的对象模型（当使用一种面向对象的设计方法时）。

- 过程视图获取设计的并发和同步方面的信息。
- 开发视图描述的是在软件开发环境中的软件静态组织。
- 物理视图描述了软件与硬件之间的映射，还反映了它在分布式方面的信息。
- 一个架构的描述是通过少许挑选使用的用例或者场景（成为一个第五视图，场景视图）来说明的。

这种架构描述框架的一个有趣的特性是使用场景视图。场景通常是基于其演示的架构上的重要需求来选择的，这允许架构师验证是否处理了这些需求。

4.5.2 Zachman 框架

由 John Zachman 创建并在“信息系统架构的一个框架”（Zachman 1987）中讨论的 Zachman 框架的目标是企业架构而不是软件架构。这种架构描述框架展示了一些值得专门讨论的特性，如图 4.9 所示。特别地，这个框架不但定义了好几个视点（列），而且明确地定义了好几个实现的层级（行）。Zachman 框架使用**视角**（perspective）这个术语来表示一个实现的层级（正如本书中描述的），使用**抽象**（abstraction）这个术语表示一个视图。

视角 \ 抽象	数据	功能	网络	人	时间	动机
范围	对业务重要的东西清单	业务执行的过程清单	业务运作所在的位置清单	对业务重要的组织清单	对业务重要的事件 / 周期清单	业务目标 / 策略清单
业务模型	例如：语义模型	例如：业务过程模型	例如：企业物流系统	例如：工作流模型	例如：主进度表	例如：业务计划
系统模型	例如：逻辑数据模型	例如：应用架构	例如：分布式系统架构	例如：人类交互架构	例如：处理结构	例如：业务规则模型
技术模型	例如：物理数据模型	例如：系统设计	例如：技术架构	例如：显示架构	例如：控制结构	例如：规则设计
详细陈述	例如：数据定义	例如：程序	例如：网络架构	例如：安全架构	例如：定时定义	例如：规则详细说明书
机能性单元	例如：数据	例如：功能	例如：网络	例如：组织	例如：进度表	例如：策略

图 4.9 Zachman 框架

视点（列）分别是：

- **数据（做什么）**。这一列描述的是企业中的实体，如业务实体和关系表。
- **功能（如何做）**。这一列描述的是企业中的功能，如业务过程和软件应用的功能。
- **网络（在哪里做）**。这一列描述的是企业中的位置，如主要的业务地理位置和系统节点。
- **人（由谁做）**。这一列描述的是企业中的人。
- **时间（什么时候做）**。这一列描述的是企业中时间的影响，如承受能力的时间安排。

- **动机（为什么做）**。这一列描述的是企业的动机，如目的和目标。

实现的层级（行）分别是：

- **范围**。这一行定义上下文视角。
- **业务模型**。这一行定义一个概念上的视角。
- **系统模型**。这一行定义一个逻辑视角。
- **技术模型**。这一行定义一个物理视角。
- **详细陈述**。这一行定义能实现的模块。
- **机能性单元**。这一行定义运作的系统。

4.5.3 Rozanski 和 Woods 框架

在《Software Systems Architecture：Working with Stakeholders Using Viewpoints and Perspectives》（Rozanski 2005）中，除了几个基础视点之外，Nick Rozanski 和 Eoin Woods 还着重讨论了一组交叉视点（我们称之为视角），如图 4.10 所示。

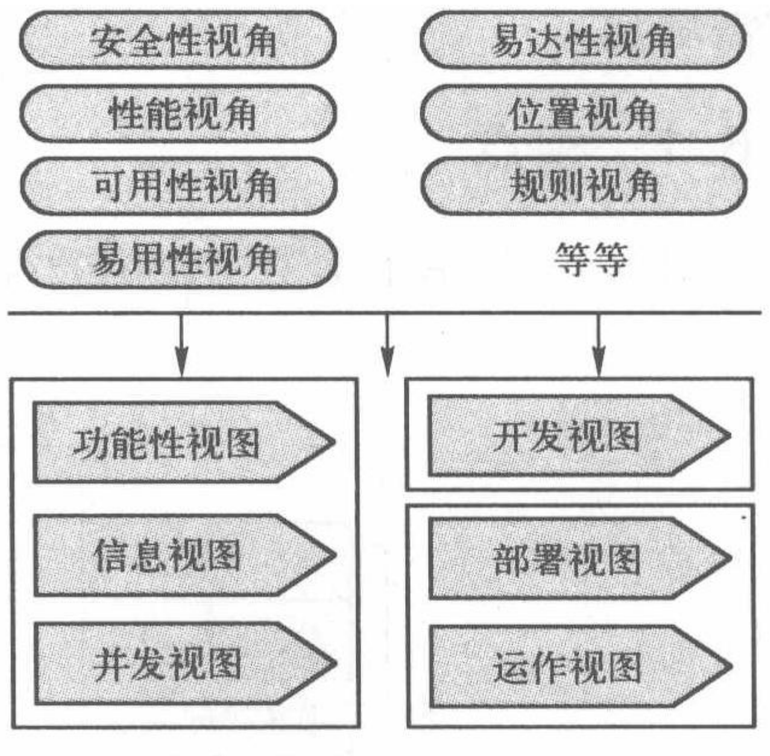


图 4.10 Rozanski 和 Woods 框架

作者们以目录的形式描述他们的不同视点。正如在他们的书中描述的，基础视点有：

- **功能性**。描述系统的功能性元素和它们的责任、界面和主要的相互作用。
- **信息**。描述架构储存、操作、管理和发布信息的方式。
- **并发**。描述系统的并发结构，还把功能性元素映射到并发单元，以便清晰地识别可以并发执行的部分以及调整和控制这些并发执行的方式。
- **开发**。描述支持软件开发过程的架构。
- **部署**。描述系统将部署的环境，包括获取系统在其运行期环境中的依赖性。
- **运行**。描述当系统在生产环境中运行时操作、管理和支持的方式。

被描述的这些视角（虽然图 4.10 中没有包含所有的视角）是：

- **安全性**。系统控制、监控和可靠地审核谁可以对什么资源执行什么动作的能力以及探测并安全地从故障中恢复的能力。
- **性能（和可量测性）**。系统在它规定的性能范围内可预测运行的能力和处理逐渐增加的工作量的能力。
- **可用性（和顺应力）**。系统完全或部分地按要求运行的能力以及系统有效处理可能影响系统可用性的失败的能力。
- **发展**。系统在面对部署之后所有系统都必然经历的变化时的灵活能力，对提供这种适应性的成本进行权衡。

- **易达性**。系统供残疾人使用的能力。
- **开发资源**。系统在众所周知的关于人、预算、时间和材料的约束之内进行设计、构建、部署和运行的能力。
- **国际化**。系统不受限于任何特定的语言、国家或文化团体的能力。
- **位置**。系统克服由其元素的绝对位置和它们之间的距离所带来的问题的能力。
- **法律法规**。系统符合当地和国际的法律、准法律法规、公司的政策和其他的规则和标准的能力。
- **可用性**。与系统交互的人可以有效地进行工作的容易程度。

4.6 一个架构描述框架

本书中使用的架构描述框架是建立在这一章中讨论的几个架构描述框架的基础之上的。这种架构描述框架可以由不同系统的架构重用，而使用这个框架的每个架构师都应该根据考虑中系统的特性来选择适当的视点。另外，应该把这个框架看成是可扩展的，因为视点可以按需增加。

4.6.1 视点

图4.11显示了本书中使用的视点（和视图）的概要信息。

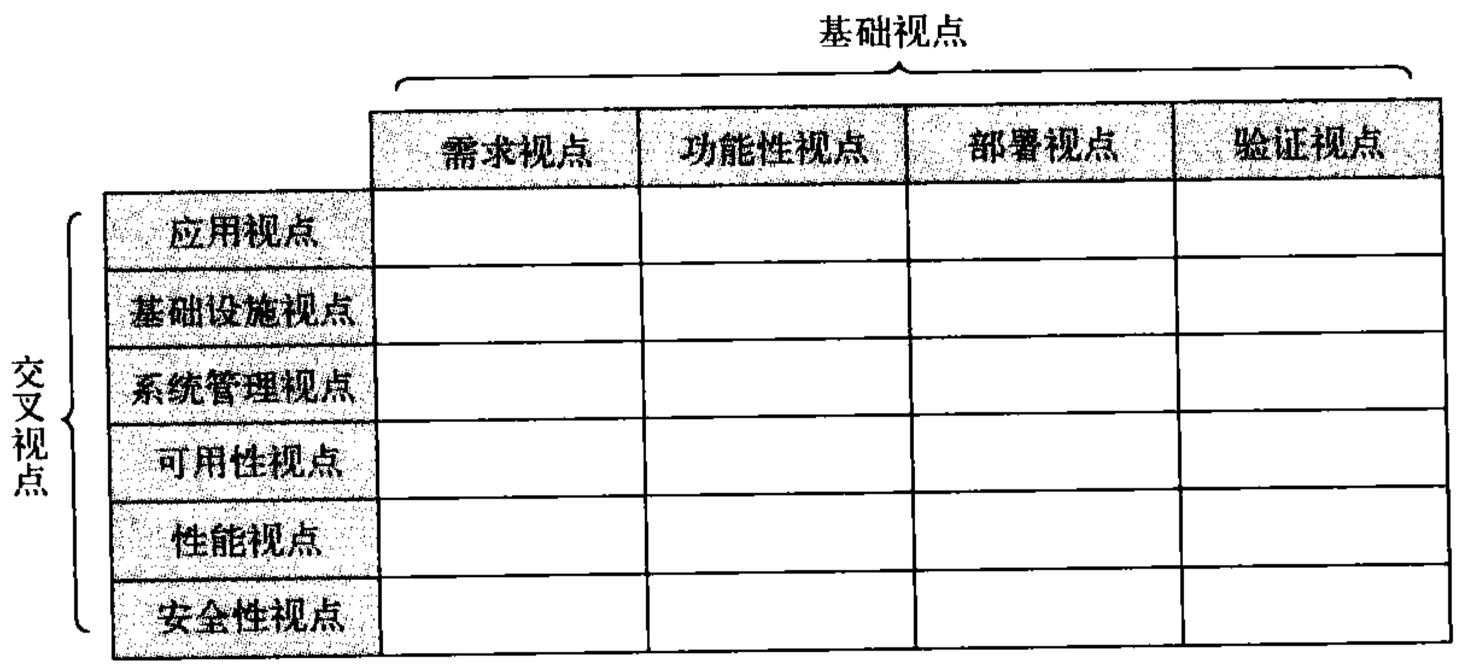


图4.11 架构描述框架

表4.1提供了这种架构的利益相关者的概要信息，并识别出可能对每个视点感兴趣的人。这张表还指出了利益相关者的范围，内部的利益相关者是当前的开发团队的成员，而外部的利益相关者不是那个团队的成员。

表4.1 利益相关者概述

角　色	范　围	主要职责
应用程序拥有者	外部	委托并为系统付费
业务管理员	外部	管理运行系统中与业务相关的元素

（续）

角　色	范　围	主要职责
配置经理	内部	定义配置管理知识库和其中元素的结构
开发人员	内部	进行详细设计并实现系统
维护人员	外部	管理系统部署后进行的更新
项目经理	内部	在项目的计划编制、人员、监控和风险方面进行管理
供应商	外部	提供系统开发或运行期间所需的软件或硬件
支持人员	外部	提供系统支持和履行诊断职能
系统管理员	外部	管理系统操作的运行
测试人员	内部	测试系统以确保它符合目标
使用人员	外部	使用系统的业务功能

表 4.2 概述了这些视点。要了解每个视点更完整的描述，请参考附录 B。

表 4.2　视点概述

视　点	利益相关者	描　述
需求	所有利益相关者	这个视点关注那些形成架构的系统需求，包括功能性需求、质量和约束
功能	所有利益相关者	这个视点关注那些支持系统功能性（逻辑上的）的架构元素，包括构建系统的结构元素、它们之间的关系和它们的行为
部署	配置经理、开发人员、维护人员、项目经理、供应商、支持人员、系统管理员、测试人员	这个视点关注那些支持系统发布的架构元素，包括像位置、节点、设备和它们之间的连接这样的元素
验证	应用程序拥有者、项目经理、测试人员	这个视点关注评估系统是否会提供必需的功能、达到要求的质量及适应定义的约束
应用	业务管理员、开发人员、维护人员、供应商、系统管理员、测试人员	这个视点关注那些提供解决方案中特定应用行为（就是系统必须提供以满足它的业务需求的行为）的架构元素
基础结构	开发人员、维护人员、供应商、系统管理员、测试人员	这个视点关注那些提供解决方案中与应用无关的行为（就是系统必须提供以支持系统业务为目的的行为）的架构元素
系统管理	开发人员、维护人员、供应商、系统管理员、测试人员	这个视点关注那些当系统部署到生产环境时有助于其操作运行的架构元素
可用性	开发人员、维护人员、供应商、系统管理员、测试人员	这个视点关注那些使系统能够达到指定可用性需求的架构元素。这个视点还关注那些使系统从故障（使系统全部或部分不可用）中恢复的架构元素
性能	开发人员、维护人员、供应商、系统管理员、测试人员	这个视点关注那些使系统能够达到指定性能需求（如响应时间和生产能力）的架构元素
安全	开发人员、维护人员、供应商、系统管理员、测试人员	这个视点关注那些使系统能够满足指定安全需求的架构元素，如授权访问系统的功能和资源。这个视点还关注那些使系统能够在系统的安全机制内发现故障并从故障中恢复的架构元素

确保在任何特定的情况下应用正确的视点，这非常重要。这里介绍的这种架构描述框架仅仅是演示这样一个架构所包含的内在原理的一个例子。

因此，对于任意给定的系统，您都可以按需要删除或者增加视图。例如，如果您正在构建一个始终在单个节点上运行的系统，那么就可以忽略部署视图（该视图考虑系统的分布）。相反地，如果您需要特别强调那些不是您所选架构描述所内在考虑的信息，那也可以选择增加视图。例如，如果系统演化是这种架构的一个主要关注点，那就增加一个演化视图。

4.6.2 工作产品

在这一章前面的“模型”部分，我们声明一个视图引用包含在一个或多个模型（工作产品）中的元素，而一个模型可能参与一个或多个视图。图 4.12 列出了下面这些工作产品，我们在本书使用的架构描述框架中使用了这些工作产品：

- 需求视图引用包含在**企业实体模型**、**业务过程模型**、**业务规则**、**变更请求**、**企业架构规则**、**现有 IT 环境**、**功能性需求**、**术语表**、**非功能性需求**、**排定优先级的需求列表**、**利益相关者请求**、**系统背景和愿景**这些工作产品中的元素。
- 功能性视图引用包含在**架构决策**、**架构概览**、**数据模型**和**功能性模型**这些工作产品中的元素。
- 部署视图引用包含在**架构决策**、**架构概览**和**部署模型**这些工作产品中的元素。
- 验证视图引用包含在**架构评估**、**架构概念证明**、**问题清单**、**RAID 日志**和**检查记录**这些工作产品中的元素。

交叉视图潜在地参考了所有这些工作产品中的元素。

	需求视点	功能性视点	部署视点	验证视点
应用视点	业务实体模型 业务流程模型 业务规则 变更请求 企业架构规则 现有 IT 环境 功能性需求 术语表 非功能性需求 排定优先级的需求列表 利益相关者请求 系统背景 愿景	架构决策 架构概览 数据模型 功能性模型	架构决策 架构概览 部署模型	架构评估 架构概念证明 （风险、假设、问题和依赖条件）日志 复审日志
基础结构视点				
系统管理视点				
可用性视点				
性能视点				
安全视点				

图 4.12 视点和工作产品

4.6.3 实现的层级

本书中使用的这种架构描述框架还合并了不同的实现层级，如图 4.13 中所示。要注意的一项重要内容是，这个框架既用于描述逻辑架构又用于描述物理架构。逻辑架构在第 8 章中详细讨论，物理架构在第 9 章中详细讨论。

逻辑架构

	需求视点	功能性视点	部署视点	验证视点
应用视点 基础结构视点 系统管理视点 可用性视点 性能视点 安全视点	业务实体模型 业务流程模型 业务规则 变更请求 企业架构规则 现有 IT 环境 功能性需求 术语表 非功能性需求 排定优先级的需求列表 利益相关者请求 系统背景 愿景	架构决策 架构概览 数据模型 功能性模型	架构决策 架构概览 部署模型	架构评估 架构概念证明 （风险、假设、问题和依赖条件）日志 复审日志

物理架构

	需求视点	功能性视点	部署视点	验证视点
应用视点 基础结构视点 系统管理视点 可用性视点 性能视点 安全视点	业务实体模型 业务流程模型 业务规则 变更请求 企业架构规则 现有 IT 环境 功能性需求 术语表 非功能性需求 排定优先级的需求列表 利益相关者请求 系统背景 愿景	架构决策 架构概览 数据模型 功能性模型	架构决策 架构概览 部署模型	架构评估 架构概念证明 （风险、假设、问题和依赖条件）日志 复审日志

图 4.13　实现的层级

4.6.4　视图一致

虽然您可能把视图看作是独立的，但其实并不总是这样。例如，您可能想坚持这样一个规则：在功能性视图中描述的所有组件都部署到部署视图中的节点上。**视图一致**这个术语就用于描述这样的关系。

4.7　软件架构文档

> **软件架构文档**通过许多不同的架构视图来描述系统的不同方面，提供了一个全面的系统架构概览。（PUR 2008）

正如我们在第 3 章中提到的，软件流程工程元模型规范（SPEM）定义了三种类型的工作产品：工件（artifact）、可交付物及成果。可交付物是指提供给内部人员或者外部的第三方人员的工作产品。**软件架构文档**是可交付物的一个例子，它是编写软件和交流软件架构的主要工具。正如第 3 章中讨论的，**软件架构文档**主要是参考其他工作产品，而不是重做这些工作产品

中定义的内容。

正如人们所预料的，**软件架构文档**这个交付物符合选择的架构描述框架。符合我们在这一章中详细阐述的这种架构描述框架的**软件架构文档**的大纲如下所示：

- 前页（扉页、变更历史、目录、图形列表、参考书目）
- 软件架构文档的目标
- 架构概览
- 架构决策
- 需求视图
- 功能性视图
- 部署视图
- 验证视图
- 应用视图
- 基础结构视图
- 系统管理视图
- 可用性视图
- 性能视图
- 安全性视图
- 附录

4.8 总结

这一章讨论了编写软件架构文档的许多基本概念，包括视图、视点以及它们与底层的工作产品的关系。我们还介绍了一个经编写软件架构文档实践证实的架构描述框架。

这种架构描述框架会在案例研究相关的章节中（第6~9章）作为范例。首先，我们略微谈一下构建一个软件密集系统的另外一个基本概念：现有资源的重用。这是第5章的主题。

第5章

可重用架构资源

软件架构师的生活是对部分在黑暗中做出的不理想的设计决策进行长期和快速的演替（Kruchten 1999）。

这一章通过讨论成功软件架构的可重用资源的使用这个关键特性来给黑暗带来一些光明。可重用资源为利用其他成功架构师的成果（从小粒度的设计模式到大粒度的已打包的应用程序）提供了一个有用的工具。

然而，对可重用资源的思考本身可能就是一个雷区。不但有许多类型的资源需要考虑，而且每个资源意味着什么及它所提供的价值并不总是清楚的。架构类型和参考架构之间的区别是什么？机制是如何区别于框架的？这一章的目的是讨论对架构师有用的不同类型的可重用资源，以及它们的特性和用法。

5.1 架构的来源

获得一个架构的灵感来自许多地方，而且随许多因素而变化，包括系统的新颖性、遵循的方法和架构师的技巧。这在“妈咪，软件架构来自哪里?”（Kruchten 1995-2）中进行了简短的描述，在其中，作者提出了架构的三个主要来源：拿取、方法以及直觉，如图5.1所示。

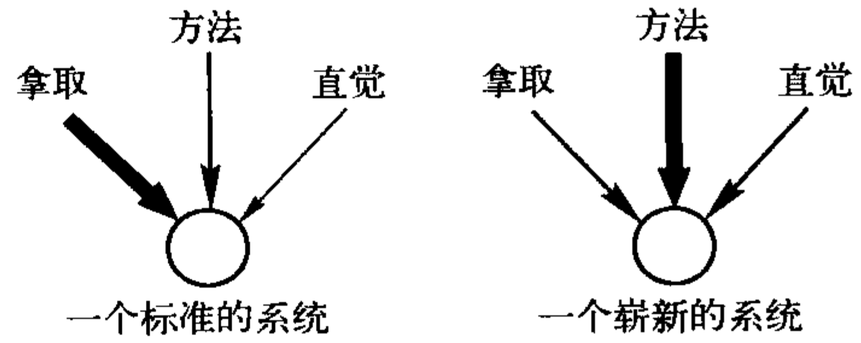


图5.1 架构的来源，来自Kruchten

就拿取而言，一个软件架构的大多数元素来自先前一个相同类型的系统、拥有完全类似特性的另一个系统、或者在技术文献中找到的一个架构——换句话说，就是利用现有的资源。方法是指一个系统化的方式，通过它来根据系统需求产生架构。最后，直觉反映了软件架构师的经验，他认可某一模式或找到一个架构元素的不同灵感。

在图5.1中显示的箭头的相对宽度也提示了三种来源的相对重要性，根据系统的新颖性而有所区别。标准的系统可能包含80%的拿取、19%的方法和1%的直觉，而崭新的系统可能包含30%的拿取、50%的方法和20%的直觉。在这两种情况下，现有资源的重用都很重要。换句话说，一个好的架构师不会做无用功。

5.2 架构资源元模型

毫不奇怪的是，成功的架构师通常是那些知道可利用资源的人。对可重用资源的考虑能够

在工作中有效地帮助架构师，因为它减少了架构师必须关注的事情的数量，他们不必做无用功。从一个项目的角度来看，项目里面的可重用资源会对项目的时间计划、成本和交付系统的质量产生重大的影响。

一般而言，可重用资源可以应用到所有软件工程方法中。一个可重用资源可以代表一个可重用需求（在不同的系统里反复出现的需求）、可重用的解决方案元素（如一个架构的模式或者可重用代码）、可重用测试、可重用的方法等。在本书中，我们的关注点在那些在定义架构中对软件架构师最实用的可重用资源上。请记住，这里讨论的资源是基于作者们与客户们一起合作的经验之上的。行业中对其中的好几个资源还没有达成一致意见，因此，我们提供的是基于我们经验的解释。

当设法为某一特定的情形寻找资源时，理解每种类型的资源的特征和资源之间的关系能帮助您理解它们的适用性。为了帮助您形成这种理解，图 5.2 列出了部分的资源元模型，我们用它来描述不同类型的资源和它们之间的关系。从这幅图中您可以进一步看到，架构资源可以分成开发期资源（如设计模式）和运行期资源（如打包应用程序）。如果想要了解更多的信息，请查看后面的“概念：开发期和运行期资源”部分。

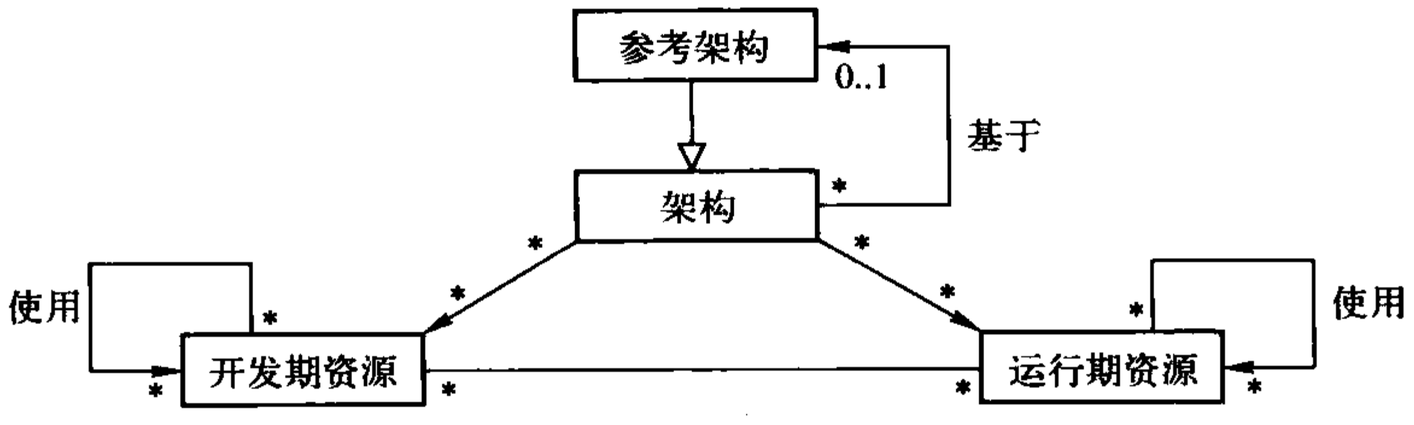


图 5.2 架构资源元模型

图 5.2 告诉您：

- 架构可能基于参考的架构。
- 架构有许多关联的开发期资源。
- 架构有许多关联的运行期资源。
- 参考架构也是一种架构。
- 参考架构可以作为许多架构的基础。
- 开发期资源可能用于许多架构中。
- 开发期资源可能使用许多其他的开发期资源。
- 开发期资源可能影响许多运行期资源。
- 运行期资源可能用于许多架构中。
- 运行期资源可能使用许多其他的运行期资源。
- 运行期资源可能影响许多开发期资源。

概念：开发期和运行期资源

这一章中讲述的资源元模型把大部分资源归类为开发期资源或者运行期资源。一般而言，这种分类方式是基于联接的标准之上的。开发期资源没有具体的运行期实现，而运行期资源有开发期的实现。

然而，几乎所有的资源对开发期和运行期都有影响。一个设计模式可能没有具体的实现，但是，运行期元素和它们的交互会遵循这个模式。同样地，一个打包的应用程序拥有具体的实现，但是，在开发期的所有架构模型中，您仍然必须考虑这个元素。

5.2.1 开发期资源

开发期资源的元模型如图 5.3 所示。

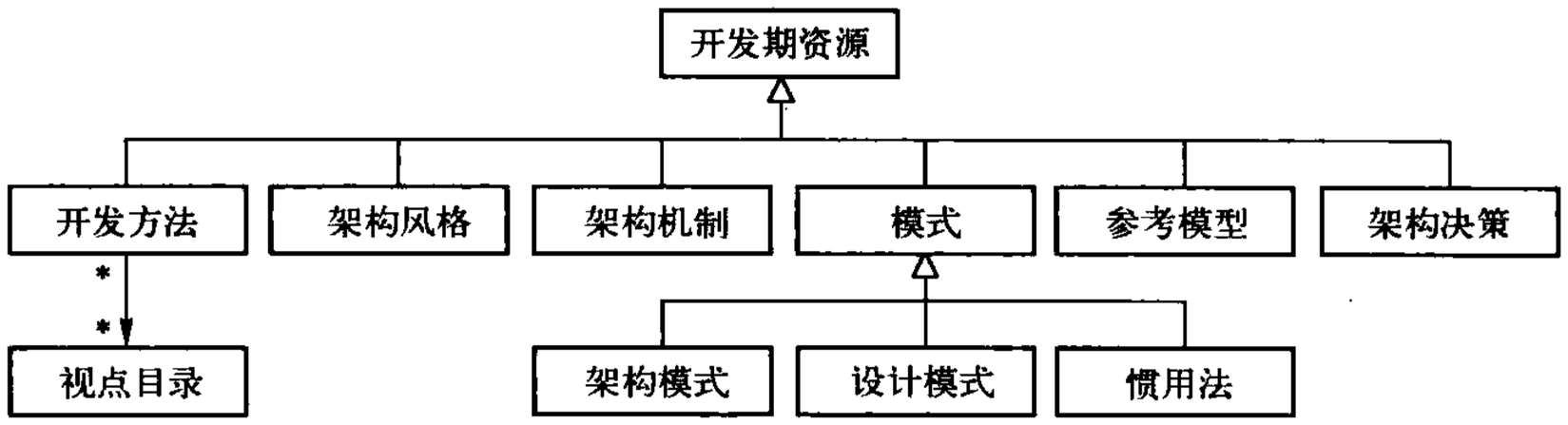


图 5.3 开发期资源的元模型

图 5.3 告诉您：

- 开发方法、架构风格、架构机制、模式、参考模型和架构决策是各种各样的开发期资源。
- 开发方法与许多视点目录相关。
- 视点目录与许多开发方法相关。
- 架构模式、设计模式和惯用法是各种各样的模式。

5.2.2 运行期资源

运行期资源的元模型如图 5.4 所示。

图 5.4 告诉您：

- 现有的应用程序、应用程序框架和组件库是各种各样的运行期资源。
- 打包的运用程序是一种现有的应用程序。
- 组件库包含许多组件。
- 一个组件可能被包含在许多组件库中。

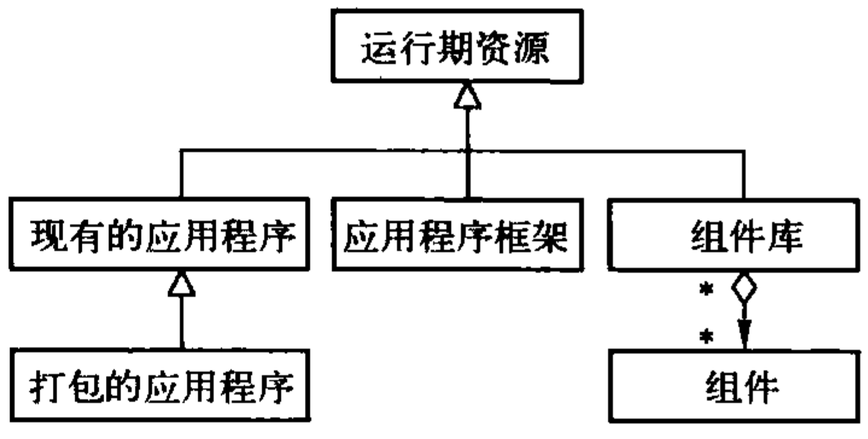


图 5.4 运行期资源的元模型

5.3 资源类型

这一节包含了在先前章节中描述的元模型中的每个资源的详细描述，我们将按顺序对它们

进行介绍。

5.3.1 参考架构

有一种特别的资源可以对架构产生重要的影响，那就是参考架构。参考架构的思想是它能为构建一个新系统提供一个初始的出发点。与其他一些可重用资源相比，这种类型的资源是相对大粒度的。

参考架构与特定的兴趣领域相关，它通常包括许多不同的架构模式，应用在它的结构的不同范围。在 1999 年，IBM 发布了一系列描述不同业务领域（如电子商务、生命科学和无线领域）的系统架构的技术参考架构。

因为参考架构本身也是一种架构，所以，发现参考架构也用与任何其他架构完全相同的方式进行描述时请不要感到诧异。它可能用一个引用了好几个其他架构工作产品（如描述架构不同方面的架构模型）的**软件架构文档**来进行描述。有时候，参考架构也可能包括具体的实现元素。当一个参考架构作为具体实现的参考例子时，该参考架构是最成功的。

5.3.2 开发方法

正如我们在本书中详尽讨论的那样，一个开发方法整理了最佳实践和相关的指导、技术和标准，通常还包含与创建工作产品相关的可重用元素，如模板和样例。同样地，开发方法本身也是一种受架构师支配的可重用资源。正如其他资源一样，应用适当的标准以便尽可能地重用，这很有意义。另外，从开发方法的角度来看，对软件流程工程元模型规范（SPEM）对象管理组织（OMG）标准（SPEM 2007）（在第 3 章中讨论的）进行考虑也是正当的。

开发方法只是一个完整开发环境的一部分，该环境的所有方面都是受架构师支配的潜在可重用资源。例如，开发方法通过某些使其自动化的适当工具（如建模工具、编译器和调试器）而增强。

5.3.3 视点目录

在这一章中我们讨论那些与软件架构师关系最密切的可重用资源，因此，我们应该明确地讲到与开发方法相关的一个特定标准：视点目录。正如我们在第 4 章中讨论的，架构师通常使用他们从视点目录中挑选且根据他们的需求进行调整的一组视点来描述他们的架构。每个视点处理利益相关者的一组相关关注点。

5.3.4 架构风格

架构风格的概念由 Mary Shaw 和 David Garlan 在他们所著的《Software Architecture：Perspective on an Emerging Discipline》（Shaw 1996）中进行了讨论。

> **架构风格**根据结构组织的模式定义系统种类。更具体地说，架构风格定义组件和连接器类型的词汇及它们如何进行组合的一组约束。（Shaw 1996）

架构风格整体应用于系统，因此，它对架构有很大的影响。另外，一个系统可能存在不止一种架构风格。正如在第 2 章中谈到的，面向服务的架构（SOA）也可以看作是一种架构风格。在《Software Architecture：Perspective on an Emerging Discipline》（Shaw 1996）和《Software Systems Architecture：Working with Stakeholders Using Viewpoints and Perspectives》（Rozanski 2005）中讲述的其他一些架构风格的例子是：

- **客户端 - 服务器（Client-server）**。这个广泛使用的风格支持客户端处理（在用户的工作站上）和服务端处理（在一个远程的服务器上，例如访问一个数据库）分离（分布式）。
- **基于事件（Event-based）**。这个风格与观察者模式（Observer）相似（在这一章后面的“模式”部分讨论），在这个模式中，它提出一个发布 - 订阅的工作方式，在这个方式中，一个或多个观察者订阅系统中发生的特定事件，当这个事件发生时，一个对象就发布一个通知。然而，这个风格没有局限于架构内的特定实例，从战略上它可以应用于架构的很大范围中。
- **管道和过滤器**。这个风格由一系列提供数据转换的过滤器和连接这些过滤器的管道组成。过滤器风格的例子包括编译器、信号处理和使用于金融市场的直通式（STP，Straight Through Processing）解决方案的事件元素，在直通式解决方案中，资本市场的交易过程和支付业务都电子化操作。这个概念也被用于能源部门的电、油和气的贸易中。

5.3.5 架构机制

架构师经常把解决方案元素描述为机制，比如“获得 X 的机制”或者“这个元素由机制 Y 支撑”。机制的例子包括持续机制、错误记录机制、通信机制和购物车。和架构风格一样，机制也可能包含一个或更多模式。

> **架构机制**代表经常遇到的问题的共同的具体解决方法。它们可能是结构的模式、行为的模式或者同时包含这两方面。（SPEM 2007）

这一章中介绍的所有术语中，架构机制可能是定义得最不充分的，因为它仅仅是您可能遇到的许多不同（和更精确）类型的架构元素的一个全能的术语。

5.3.6 模式

术语“模式”用来表示许多类型的可以构思的可重用元素，因此，它代表类型非常普通的资源。

> **模式**是在给定的上下文中针对一个常规问题的一个常规解决方法。（UML User Guide 1999）

因为模式是如此常规的一个概念，所以它能应用在贯穿开发生命周期的许多方面，从而形成需求模式、架构模式、设计模式、编程模式（也称为习语）、测试模式、项目管理模式、方

法模式、组织模式等。

基于模式对于架构师的重要性，把重点放在识别模式和编写模式文档的技巧上就一点都不令人惊讶了。应用从土木工程中学到的经验，模式运动在过去的十多年迅速崛起。许多出版物和网站致力于各种类型的模式，包括与平台无关的设计模式（Fowler 1997）、架构模式（Buschmann 1996）和设计模式（Gamma 1995）。这些出版物的灵感起源于建筑架构师 Christopher Alexander 的那些有影响力的作品（Alexander 1964、Alexander 1977 和 Alexander 1979）。甚至有反模式的资料（Brown 1998）。

> **反模式**是产生截然相反结果的一般的事故模式或解决方法。反模式可能是一个位于错误的上下文中的模式。（Brown 1998）

架构师对那些直接影响他们工作的模式特别感兴趣，也对那些影响详细设计和实现的模式特别感兴趣。通常包括下列模式：

- **架构模式**。这些模式是粗粒度的，在《Pattern-Oriented Software Architecture：A System of Patterns》（Buschmann 1996）中定义的层（Layer）模式就是一个例子。这个模式把应用程序按组组织起来，其中每个组都代表一个特定的抽象层级。分层的最熟悉的例子之一是由国际标准化组织（ISO）定义的 OSI 七层模型，在这个模型中，每一层都建立在更低层的能力上。
- **设计模式**。这些模式是较细粒度的，在《Design Patterns：Elements of Reusable Object-Oriented Software》（Gamma 1995）中定义的观察者模式就是一个例子。这个模式也就是大家熟知的发布 - 订阅模式，该模式容许一个对象订阅一个目标对象中发生的事件并当这些事件发生时获得提醒。架构师经常确定这些设计模式用于一个项目，因为这样的决策对设计者的工作进行了约束，最终导致一定的连贯性及架构完整性。
- **习语（编程模式）**。这些模式是特定编程语言中定义的可重用编程表达式。架构师通常不涉及习语的识别，虽然它们可能影响某些方面，例如影响编程异常出现及处理的方式。

在重用现有经验方面，模式是架构师的关注点，它们是用于定义应该始终如一应用的解决方案的元素的有价值的机制，它们还为架构师提供机会以系统化的形式共享他们关于项目的知识。尽管存在许多描述模式（或反模式）的模板，但是，它们有许多共同点。大部分的模式描述包括：

- 模式名称
- 模式使用的上下文
- 模式解决的特定问题
- 模式提供的解决方案
- 与这个模式相关的模式

模式解决方案在结构和行为方面都可以可视地描绘。图 5.5 是一个显示观察者模式中参与者的统一建模语言（UML）类图，它来自于《Design Patterns：Elements of Reusable Object-Ori-

ented Software》（Gamma 1995）。这个模式的主要目标之一是使对象不需要知道观察它的元素。在这个模式中，通过引入一个维护一系列观察者的超类（Subject）来实现这个目标。

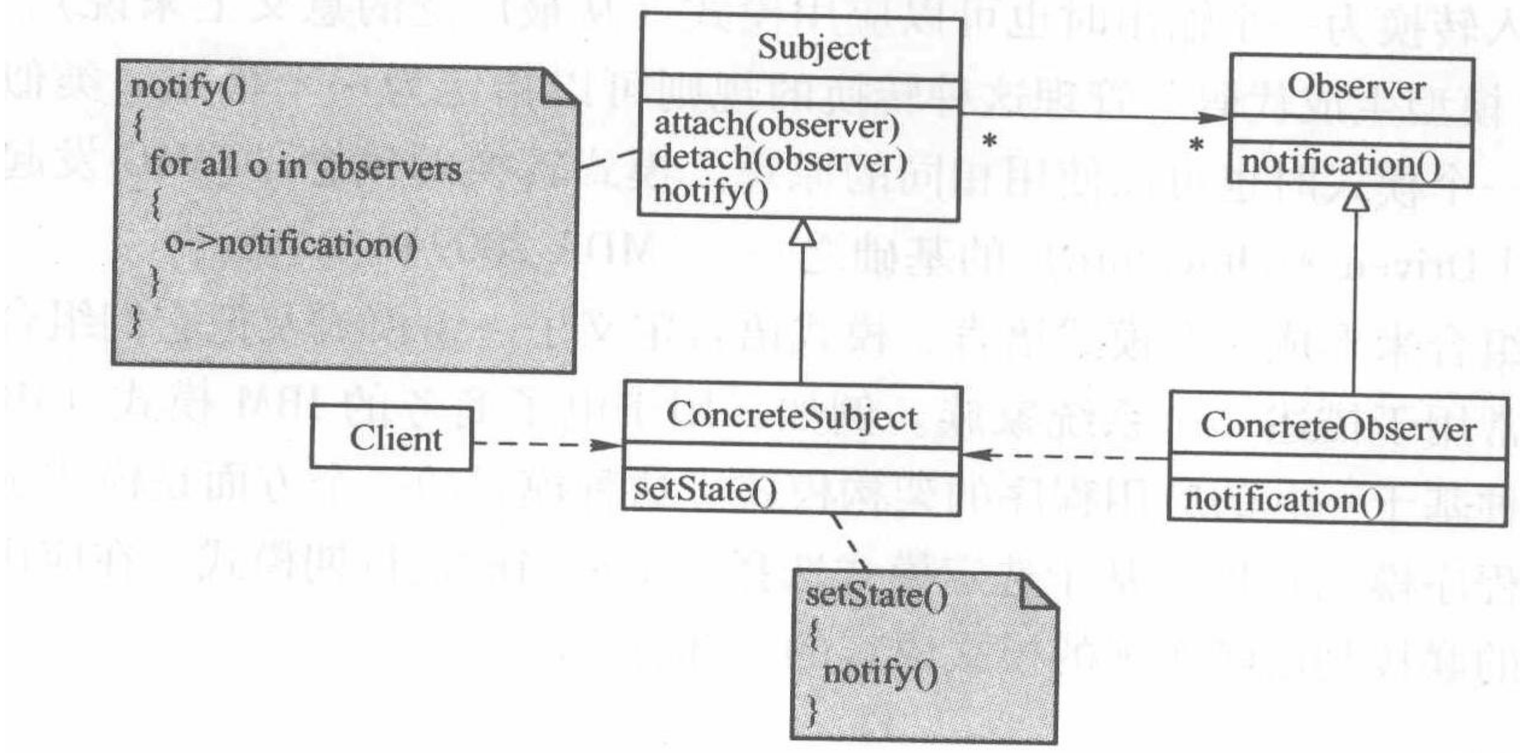


图 5.5　观察者模式中的结构元素

既然已经显示了一个模式的结构，那就再显示一下它的行为。图 5.6 是一个显示观察者模式内的结构元素如何协作的 UML 时序图。这个图演示了当对象更新时的行为。这个更新导致一个提醒消息发送给观察者，通知他们该对象已经更新。

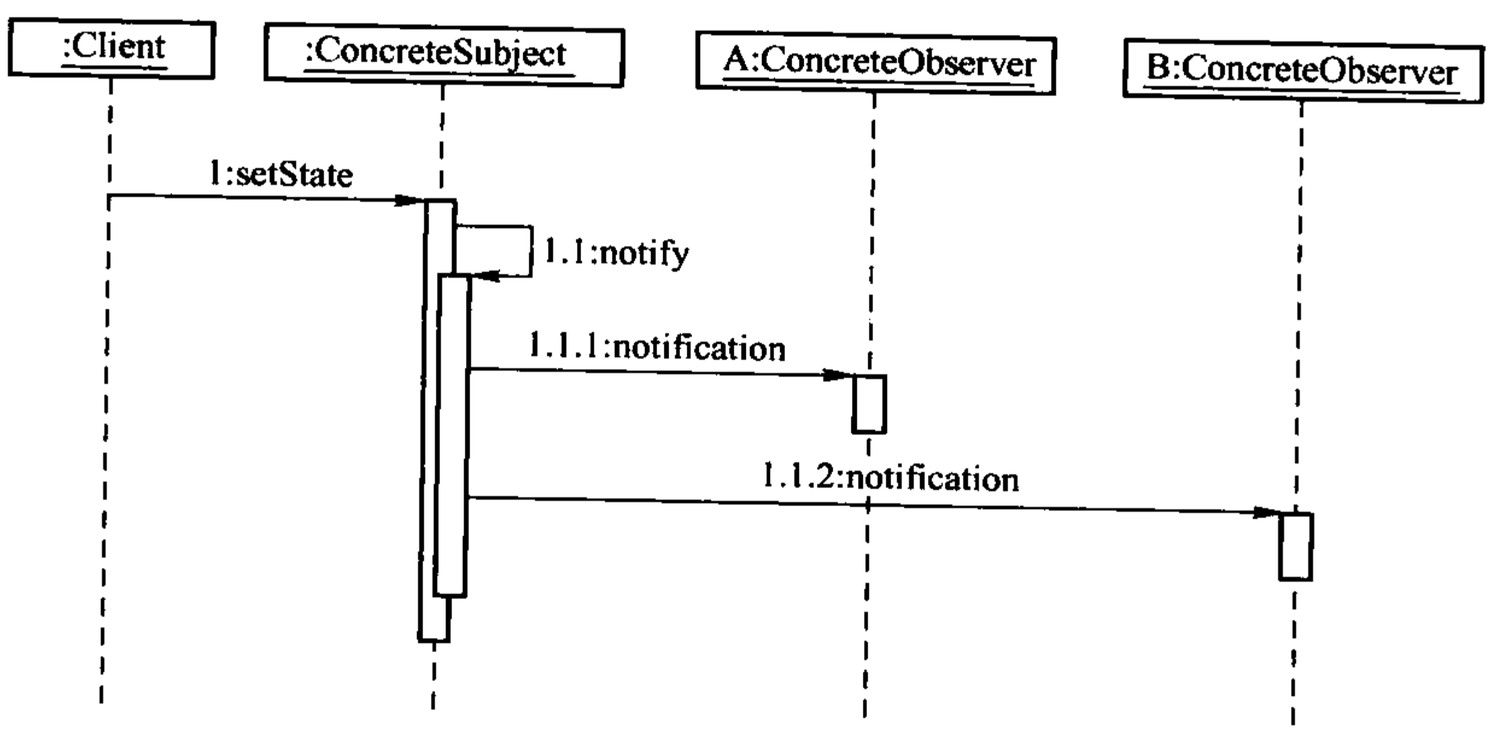


图 5.6　观察者模式中的行为

这里的要点不是简单地演示观察者模式，而是演示如何获取和交流架构的经验。架构师通常重用外部资源确定的模式，也定义他们自己的模式用于项目，从而确保一定的架构连贯性。当然，获取以前的一个模式并注明它曾经应用在什么地方肯定比详细描述这个模式的每个用途更有效。

在一个 UML 建模工具中描述一个模式也为把这个模式自动应用到现有的模式元素中提供了机会。然后，您可以确定那些满足模式内特定角色的模型元素并通过把这个模式应用到相应的地方来（自动地）更新这个模式。在我们的观察者模式例子中，任何确认为观察者的元素

都会增加一个提醒的操作。一个模式在建模工具中的应用可以更广泛，可以创建属性、操作、关系甚至新的建模元素。这种能力支持一个模式驱动的开发方式转到软件开发。

当把一个输入转换为一个输出时也可以应用模式（从最广泛的意义上来说）。例如，您可以根据一个 UML 模型生成代码。管理这种转换的规则可以指定为一个模式。类似地，当从一个模式转换为另一个模式时也可以使用相同的原理。模式转换的概念是 OMG 发起模型驱动架构（MDA，Model Driven Architecture）的基础之一。（MDA 2009）

模式也可以组合来形成一个模式语言，模式语言定义了一组模式及把它们组合在一起的规则。模式语言经常用于描述一个系统家族。例如，用于电子商务的 IBM 模式（IBM 2009）定义了一套描述各种基于 web 的应用程序的架构模式。这种模式的一个方面是模式选择流程（支持确定一个应用程序模式）以及基于选定模式选择一个适当的运行期模式。在应用程序模式和运行期模式之间的联接是已经实现的模式语言的一部分。

5.3.7 参考模型

参考模型是一个特定关注领域的实体、它们的关系和行为的一个抽象表示，它通常为更具体的元素的开发形成概念基础。业务模型、信息模型和术语表就是这样的例子。

参考模型的一个例子是您期望可以在一个财务机关发现的定义业务实体和业务流程的一个业务模式，因为这样的实体和流程影响支持它们的 IT 系统。对于架构师来说，在他们的工作中考虑这样的参考模型明显有帮助。业务实体通常代表那些影响不同架构层中元素的识别和命名的简洁抽象。IBM 的保险应用架构（IAA，Insurance Application Architecture）是参考模型的一个具体的例子。这个模型提供了代表保险行业普遍使用的最佳实践的 200 多个业务流程和 700 个独立的业务活动。

5.3.8 架构决策

架构决策是关于一个软件系统整体或它的一个或多个核心组件的刻意设计决策。这些决策决定系统的非功能性特性和质量指标。（Zimmermann 2008）

架构师在整个项目周期中一直在作决策，决策的基本原理可能来自于经验、方法或其他资源。然而，把架构决策看作是最好的可重用资源，这有助于确保不会重走其他架构师已经走过的路而做重复发明。架构决策通常比最终的架构解决方案需要更多信息，因为它们通常需要讨论考虑的可选方案和作出选择的基本原则。

把架构决策看作是可重用资源，这是一个正在形成的领域。同样地，与这一章中讨论的其他资源相比，支持这类资源的方法和工具也相对不成熟。

5.3.9 现有的应用程序

当然，任何现有的应用程序都是高可重用性的资源。这些应用程序通常指遗留应用程序。“遗留”这个术语可能给人负面的印象，没有经验的架构师把遗留应用程序看作是要回避的东

西而不是真正的有价值的资源，这很常见。

愤世嫉俗者，是只知一切事物的价钱而不知其价值的人。

——Oscar Wilde

当您把现有应用程序合并到您的架构中时，这项工作的主要精力集中在集成而不是定制开发，重用的水平明显很高。这种集成有时候是指遗留集成或企业应用集成（EAI）。

5.3.10 封装的应用程序

封装的应用程序是提供很多能力（和重用）的一个粗粒度的商业化成熟（COTS）产品，如一个客户关系管理（CRM）应用程序或企业资源计划（ERP）应用程序。Siebel 和 SAP 就是封装应用程序的例子。封装应用程序可以部署为解决方案的一部分或被当作一个主机服务（参见后面的“概念：软件即服务”部分）。

很明显，如果封装应用程序占我们系统的很大一部分，需要的定制开发量就会显著减少，而精力会移到封装应用程序的配置和集成上。另外，许多您作为架构师需要作出的重要决策都已经确定并包含在使用的选定封装应用程序中。

概念：软件即服务

软件即服务（SaaS）是软件部署的一种方式，采用这种部署方式，应用程序将部署为通过因特网可用的主机服务。这种方式的主要优点是这些服务的消费者不用担心应用的安装、操作和支持。这种方式的主要缺点是放弃了应用某些方面的控制，如可能动摇架构稳定性的产品升级。

当采用一个封装应用程序时，关注的一个方面是这个封装应用程序在满足需求方面的适应/缺口分析。在某些情况下，当采用一个封装程序时，适应这个应用程序强加的任何约束比试图改变这个应用程序本身来得更简单。从这种意义上来说，使用封装应用程序可以对架构产生颠覆性的影响，因为是选择的技术影响了架构，而不是自上而下的需求影响了架构。

对这么重要的资源的选择也会影响到您对考虑中系统所采取的开发方式。选择一个封装系统可以导致采用与集成遗留系统或构建定制代码所不同的软件开发方法。架构师经常采用混合的方式，因为许多软件开发项目都同时使用许多类型的可重用资源。

5.3.11 应用框架

应用框架（有时候简单地称为框架）代表一个应用程序的特定领域的部分实现。应用框架在范围上可能变化很大。一些框架代表完整的平台，如 Java EE 和 .NET；而另一些框架则关注一个特定的领域（可能在一个平台的范围内），如数据访问或用户界面。举例来说，Hibernate 就是在一个 Java 环境中把对象模式映射到关系型数据库的一个框架。Java Server Face 和 ASP. NET 是支持创建用户界面的框架。

一个应用框架（就像一个模式）的关键特征之一是它明确地定义了框架中固定的内容

（在结构和行为两方面）与由使用这个框架的任何应用程序改变的内容之间的分界线。毫不奇怪，一个应用框架经常基于一个或多个模式，而且经常按类似的风格编写文档。

“应用框架”这个术语不应该与架构描述框架（在第4章中讨论的）相混淆，架构描述框架是基于一组视点来描述软件架构的一个标准。

5.3.12 组件库/组件

在绝大多数情况下，这里讨论的几个资源类型代表一个解决方案的组件，包括现有的应用程序和封装的应用程序。然而，在可重用资源的背景下，我们使用“组件”这个术语代表那些有完整实现但与完整应用程序相比粒度更细的东西。

虽然如此，这样的组件在粒度上也可以变化很大。一个组件可以代表架构中的一个重要元素，如消息代理。它也可以代表SOA中的一个服务或特定技术的一个组件，如一个Enterprise JavaBean。重用的一个广义理解形式是关于细粒度的元素，例如描绘一张表格的用户界面小部件。

组件也可以通过组件库、类库或程序库的形式来提供。Java类库就是组件库的一个例子。

5.4 架构资源的属性

正如您将看到的，资源有好几个属性可以修饰。作为一个例子，图5.7中列出了这一章先前讨论的一组架构资源，它由粒度和清晰度（实现的程度）这两个属性进行分类。当消费者设法寻找适合某一情形的资源时可以使用这些属性。

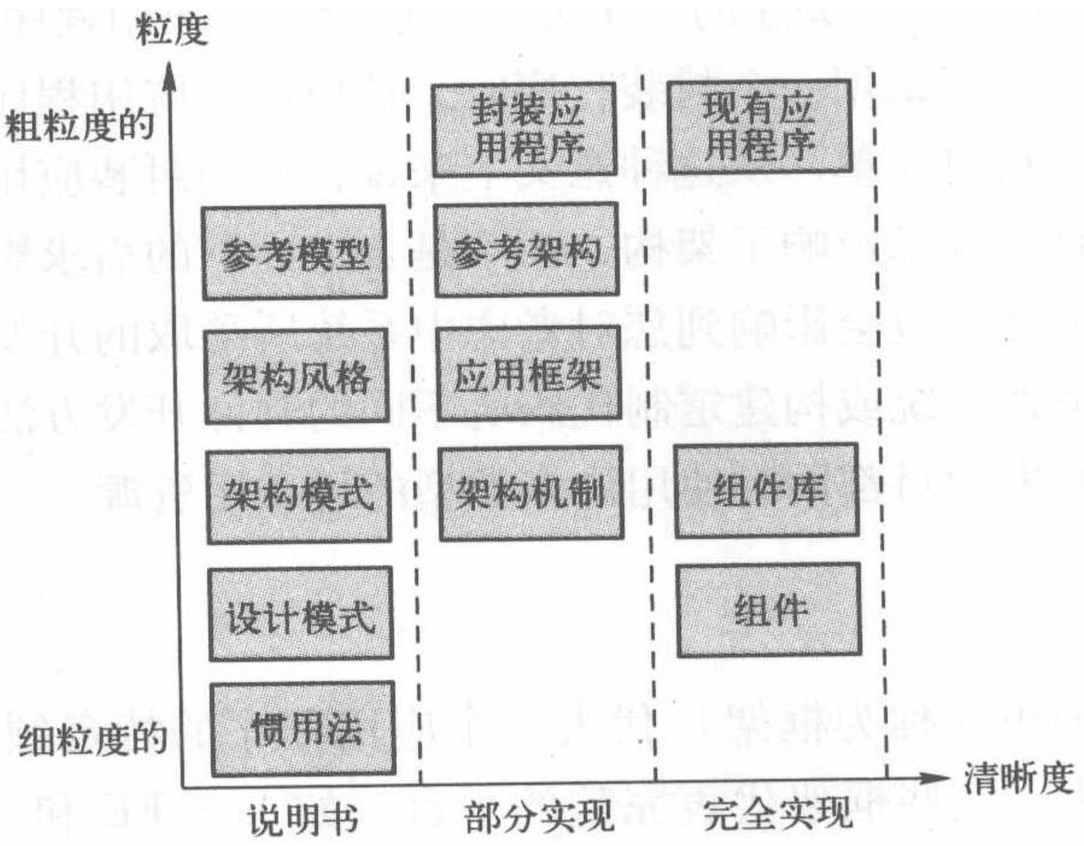


图5.7 利用属性给各种资源分类

正如图5.7所示，资源可以从它们的粒度（大小）和清晰程度（实现）来考虑。粒度既与组成这个资源的元素数量有关，又与这个资源对整个架构的影响有关。清晰度与这个资源可

以接受的完善程度有关，它可能是下列情况之一：

- **说明书**。这种资源没有实现而以一个抽象的形式（如模式或文档）呈现。这种资源包括各种类型的模式、架构风格和参考模型。
- **部分实现**。这种资源可以看作是部分实现的，但是在它们可以实际使用之前需要另外的元素。这种资源包括应用框架和封装应用程序。
- **完全实现**。这种资源可以看作是完全实现的，可以不作修改就进行使用。这种资源包括组件和现有应用程序。

这样也好，但是粒度和清晰度可能仅是感兴趣的两个属性。如果把属性分配给特定类型资源的目的是帮助潜在消费者查找资源，那就需要考虑更多的属性。与所有资源有关的一些基本属性包括：

- **清晰度**。资源提供的一个具体实现的程度
- **作者**。最初创建这个资源的人
- **解决的关注点**。资源希望解决的关注点
- **包含文档**。组成这个资源的实际文件
- **粒度**。这个资源与其他资源相比的大小指标
- **名称**。这个资源的名称
- **先决条件**。在这个资源可以使用之前必须满足的条件
- **相关资源**。与这个资源相关的任何资源及相互之间关系的本质
- **状态**。这个资源在任何资源生命周期方面的当前状态（如已提交或已审核）
- **类型**。这个资源的类型（例如参考架构或设计模式）
- **使用指导说明**。使用这个资源的指导说明
- **可变性**。这个资源的可变程度（如不可变化、有限的或大范围变化）
- **版本**。这个资源的版本号

由资源使用所产生的属性包括：

- **反馈**。与这个资源相关的任何反馈注释
- **等级**。与这个资源相关的等级指标
- **使用数**。这个资源被请求的次数指标

与资源应用背景相关的属性包括：

- **业务领域**。这个资源应用的领域（如电信或金融服务）
- **开发原则**。这个资源应用的原则（如需求或架构）
- **开发方法**。这个资源应用的方法（如 Scrum）
- **开发阶段**。这个资源应用的开发阶段（例如初始阶段）
- **开发范围**。从开发视角看到的这个资源的范围（如软件工程或企业架构）
- **技术领域**。这个资源应用的技术（而不是业务）领域（如嵌入式系统）

另外，有许多非功能性属性也与资源有关，如成本、性能、可测量性等。

5.5 重用的其他考虑因素

过去的几年已经把可重用资源看成是提高项目成绩和增加时间价值的一种方式。例如，SOA 的广泛接受已经导致人们重新关注如何识别、说明、了解、部署、管理、支持、维护、更新及（最后）撤销相关的可重用资源（在这种情况下就是服务）。在大多数情况下，在战略上有重用的考虑。不幸的是，这些主题的详细讨论本身就值得写一本书，虽然我们在第 10 章中确实讨论了一个系统之系统（system-of-systems）的方式来识别大规模的可重用资源。

描述可重用资源的方法是支持任何重用的开端，这就是需要可重用资源规范（RAS，Reusable Asset Specification）的原因（RAS 2004）。这个 OMG 标准从根本上定义了一个重用策略的两个方面：一方面是描述一个可重用资源的标准，第二方面是定义一个顺应 RAS 知识库（称为 RAS 知识库服务）的接口的标准。任何希望能够可靠地处理可重用资源的组织都应该查看 RAS 并重用实现了这个标准的知识库。

当然，要获得成功的资源重用所涉及的绝不仅是解决纯粹的技术挑战。我们没有逐一细说在组织中开始支持战略性重用所需的角色、执行的任务（尤其是关于资源的创建、使用和维护）和支持开始重用所需的特定工作产品（如资源目录）。在组织层面还有更广泛的牵连，例如，重用文化的培养及评定什么程度的重用文化对组织来说是实用的和有利的。

5.6 总结

许多类型的可重用资源都由软件架构师支配，当正确地使用这些资源时可以显著地提高项目的成绩。这一章提供了架构师在工作中经常遇到的资源的一个概览，还对资源的属性进行了讨论。

我们将在后续的案例学习章节（第 6 ~ 9 章）中考虑可重用资源的应用。

第 6 章

案 例 介 绍

本章的目的是介绍一个贯穿于第 7 ~9 章的研究案例。在这几章中，我们将介绍并逐步讨论详细需求及相应的解决方案，与架构相关的元素会作为讨论的重点。这一章将设定这个案例的背景并描述解决方案的高层次需求。它包含的信息通常可以在商业案例、项目目标文档或业务模型中找到。这些信息会包含在定义详细需求时的几个任务中，正如您将在第 7 章中看到的。

6.1 流程应用

在讨论这个案例之前，我们应该对流程应用的各个元素做个概括性的说明，对这些元素的讨论将贯穿于第 7 ~9 章。正如我们在本书最开始提到的，您将看到的是对一系列可以按多种方式应用到多种项目的实践（practice）的阐述。这部分（以及第 7 ~9 章中的相应部分）提供了如何应用这些实践的指导。

实践是解决一种或几种经常出现的问题的方法。各种实践可以看作是很多“块”流程，可以分别采用、生效和配置。

正如第 3 章“基本方法”中所述，我们可以更准确地说实践是一个方法块，因为实践具有两个维度：方法的内容和流程，两者都可配置。我们在第 3 章中说过，方法的内容描述了独立于生命周期的元素，如角色、任务和工作产品等。然后，一个流程为这些元素定义一个应用的次序，同时还考虑迭代。

从第 7 ~9 章，我们会讨论实践的每一“维”。现在我们就这一章描述的各种元素（角色、工作产品、迭代等）的应用做一些概要的说明。我们试图在非常简单的、持续几周或几个月的项目和非常复杂的、持续几年的项目之间进行平衡，这一章的内容会呈现这一点。因此，本书描述的许多种元素都需要针对您的开发项目进行适当的“剪裁”。也就是您需要考虑在多大程度上应用这些元素。为了举例说明，这个研究案例应用了本书中描述的所有元素——不多也不少。

首先需要剪裁的是组成“方法内容”的那些元素：角色、任务和工作产品。一个必然会问的问题是“每一种元素我们需要多少个?”当然，答案是“具体情况具体分析。”一个总体建议是考虑所有元素后进行剪裁。表 6.1 列出了两个虚拟的项目（一个大项目，一个小项目）作为剪裁的例子。

表 6.1 正确剪裁的例子

元 素	小 项 目	大 项 目
角色	一个人扮演**首席架构师**、**系统架构师**、**基础设施架构师**和**数据架构师**	由不同的人来担任不同的角色：**首席架构师**、**系统架构师**、**基础设施架构师**和**数据架构师**。另外，团队还包括安全架构师和性能架构师
任务	**架构概览**作为架构的骨架从新创建，然后固定，因为其在确定以后将不再更新	架构总览创建并作为正式的工作产品进行保存，在整个项目的生命周期中对其进行维护
工作产品	仅在纸上论述架构概念上的可行性（不需要构建可执行程序） 架构描述框架仅包括需求、功能和性能意见，**软件架构文档**描述与之相关的观点	将构建一个可执行程序作为架构概念上可行性的论证 架构描述框架包括需求、功能、部署、有效性、系统管理、性能和安全的意见，并加上一个信息视点用以适当强调这一特殊的方面。与之相应的意见包含在**软件架构文档**中

与此类似，对于方法中的过程元素、阶段、迭代等，也应该进行剪裁。需要注意的事项包括：

- 过程中的阶段定义和所有与之相应的里程碑。
- 基于一个迭代在整个项目生命周期中所处的位置和阶段，该迭代中架构任务的侧重点。
- 某个任务在一次迭代中执行的次数。执行多次可能非常有意义。
- 迭代中任务的次序（本书所建议的排序——仅是建议）。

它们之所以如此灵活，是因为在每个项目的初期这些元素很少能被确定。例如，在项目的进行过程中，领域和需求经常演化（甚至被发现）。随着时间的推移，架构也随之演化。随着不同任务的侧重点的变化，项目中涉及的角色也随之变化。在第 7 ~9 章，我们将进一步讨论这个话题。

另外，取决于项目的持续时间，随着项目的预算周期、组织的变化、新的商业优先级、公司的能动性等因素的确立，关键风险承担者的期望和影响力在项目周期中会发生波动，这同样可能会影响对它们的剪裁。

最佳实践（practice）：选择一组相关的视点

正如在本书第 4 章中所述，我们根据在建系统的特性来选取视点。我们可以根据简单的描述规范来添加或删除视点。这些视点可能包括：

- 数据（或信息）视点，这种视点主要关注在系统中的持久数据。本书把这些关注作为功能视点的一部分。
- 用户体验视点，这种视点重点记录用户与系统的一个或多个界面的交互。《Building Web Applications with UML, 2nd ed.》（Conallen 2003）介绍了用户体验建模。
- 并发视点，这种视点关注系统的并发和异步元素，如系统中可以执行的进程和线程。并发是《The “4 +1” View Model of Software Architecture》（Kruchten 1995）中的观点之一。
- 开发视点，这种视点描述了开发环境中软件组织的静态信息。这种视点同样也是《The “4 +1” View Model of Software Architecture》（Kruchten 1995）中的观点之一。

6.2 案例研究范围

如果本书想要介绍所有可能存在的商业挑战、商业领域、架构风格、可用技术等，那么，它将会很厚。实际上，正如第 1 章中所述，本书仅仅关注那些与架构相关的任务，这些任务经常会出现，而且可能会应用于不用的商业组织、业务领域、系统和项目中。

尽管本书所描述的流程可以用于不同目的，但是，我们还是把内容限制在一个研究案例内，通过它，我们能够举出与项目有关的关键任务的典型例子。

表 6.2 给出了这个案例研究特征的大致描述。尽管这些特征不完全适合于所有项目，但是，我们确信本书描述的任务经过适当调整后可适用于与这些任务有关的任何情况。在第 10 章中我们将讨论处理复杂系统时的关注点。表 6.2 使用了棕色地带（brownfield）和绿色地带（greenfield）这两个词汇，我们将在补充内容“概念：软件开发中的棕色地带和绿色地带”中讨论它们。

表 6.2 案例研究的特征

特　征	包含于案例中	不包含于案例中
系统的数目	一个	多个
项目的个数	一个	多个
时间范围	开发过程	运行和维护过程
业务环境	考虑业务环境	定义业务环境
开发领域	软件	软件、硬件、人员和信息
案例适用方式	信息系统	其他（例如图像处理）
架构风格	客户端 - 服务器	其他（例如事件驱动）
自由度	高（主要为绿色地带）	低（主要为棕色地带）
第三方商业软件包	用作为系统的组件	用作为系统的主要组成部分
现有系统	用作为系统的组件	用作为系统的主要组成部分
业务领域	零售业	其他（如金融系统）
技术领域	Web 技术	其他（如安全性要求很高的系统、嵌入式实时系统）
团队位置	在同一个地方	分布于各地

概念：软件开发中的棕色地带和绿色地带

在《Eating the IT Elephant：Moving from Greenfield Development to Brownfield》（Hopkins 2008）书中，作者参照建筑业，使用了**棕色地带开发**（brownfield development）和**绿色地带开发**（greenfield development）这两个词汇。在软件工程中，棕色地带总是对架构有某种限制，架构不能从零开始。

在有棕色地带的地方，现有系统使二次开发和重用变得很复杂。绿色地带则是干净的没有遗留系统的环境。（Hopkins 2008）

6.2.1 项目团队

我们在贯穿这个案例的相关章节中都提到了构成项目团队的角色。图6.1描述了这些角色和他们所遵循的定义。注意，这里的各种角色有可能由一个人扮演，有时候由一个架构团队扮演。尽管本书的每一章中都有关于如何应用任务的内容，但是，这些内容都没有假定团队的规模。

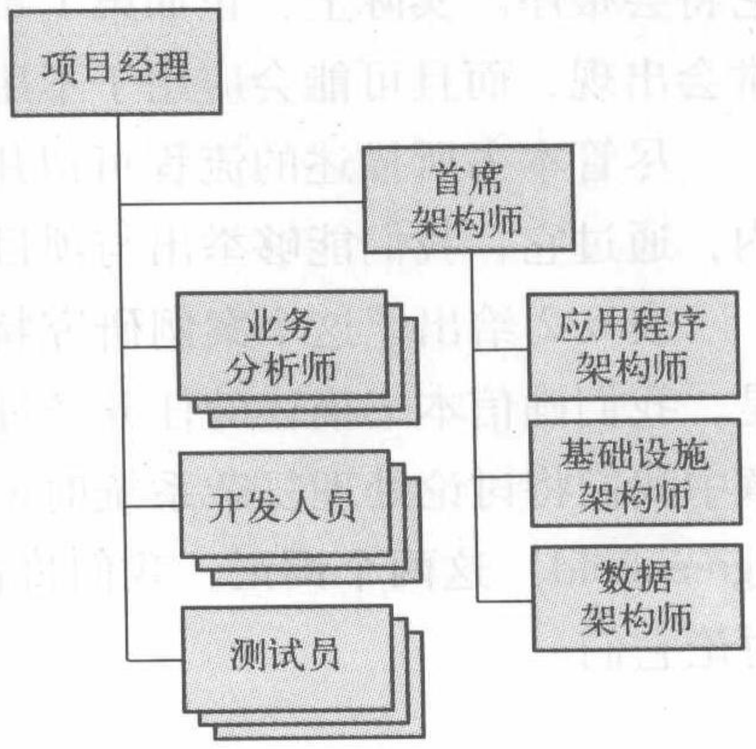


图6.1 项目团队

- **项目经理**（Project Manager）负责领导发布解决方案的团队，制定项目管理规范、工具和技术。项目经理总体上负责项目的范围、开销、进度和合同上的交付项目，当然还有其他各种问题、风险和变化等。
- **业务分析师**（Business Analyst）负责确定项目需求及编写需求文档，还从业务层面分析这些需求。扮演这个角色的人定义了当前系统和未来系统所运行的场景（流程、模型、用例和解决方案），而且还和风险承担者及架构师协作，确保业务需求能正确地转化为系统解决方案的需求。
- **开发人员**（Developer）负责系统详细设计，包括定义系统结构、实现这些详细设计、对系统进行单元测试、集成其他开发人员的工作并最终使系统运行起来。
- **测试人员**（Tester）负责发现软件的缺陷及编写测试文档，对系统质量发表总体意见，还需要确定系统的功能是否符合设计要求。测试人员在对开发规范中的单元测试进行补充的同时，还需要关注并把其他人员提供的集成测试和系统本身视为一个整体。
- **首席架构师**（Lead Architect）负责确定系统架构的总体技术方向。扮演这个角色的人还负责提供这些决定的依据、平衡不同风险承担者之间的利益、管理技术上的问题和风险，还要确保这些决定有效地表达出来、获得认可且得到遵循。
- **应用架构师**（Application Architect）关注系统如何使业务流程自动化并满足业务需求。扮演这个角色的人主要关注如何使业务需要的功能获得满足，同时考虑使系统满足非功能性需求（如质量和约束）。
- **基础设施架构师**（Infrastructure Architect）关注那些和业务功能无关的系统因素，如持久的机制、硬件和中间件。这些因素支持与系统相关的元素的运转。扮演这个角色的人关注对系统质量有重要影响的因素，进而确定需要确保满足哪些非功能性需求。
- **数据架构师**（Data Architect）关注系统的数据方面，尤其是用适当的机制存储的数据，如数据库、文件系统、内容管理系统或其他存储机制。扮演这个角色的人定义数据有

哪些合适的属性，如结构、来源、位置、完整性、可持续性（availability）、性能和有效期。

6.2.2 外部影响因素

大多数项目都不是从零开始的。现在我们考虑一下这个案例的业务和技术问题；我们将在第 10 章中更详细地讨论外在因素对项目的影响。为了方便，我们假设有一些输入信息来源于项目本身之外。我们在补充内容“最佳方法：理解更广义的上下文”中将讨论获得这些输入信息的可能性。

从业务角度来看，最主要的输入信息是对任何可以由 IT 技术（及任何可用应用程序）支持的业务方面的确认和基本原理。然而，从业务方面获得输入信息不仅限于此，下面的这些工作产品也可能作为输入：

- **业务实体模型**（Business Entity Model）。这个工作产品（或类似的工作产品，如领域模型）定义了所考虑的业务领域的关键概念。
- **业务流程模型**（Business Process Model）。这个工作产品定义了业务所进行的活动和执行这些活动的角色。
- **业务规则**（Business Rules）。这个工作产品定义了所开发系统必须满足的任何和业务有关的规定（policies）和条件。

从更技术性的角度来看，您或许会接触到下面这些或类似的工作产品。这些工作产品有助于指导和限制这个组织中创建的任何系统，而不是针对某一特定的系统。

- **企业架构规范**（Enterprise Architect Principles）。这个工作产品包含一些准则（rules）、标准（Standards）和指导（guidelines），它们影响所有企业应用的架构的创建方式。它可能包含一个得到认可的供应商列表或一个像“购买与构建的比较”这样的规定。这个工作产品将在企业的层面进行定义（作为企业架构的一部分，正如在第 10 章中所述）并应用到开发的所有系统中。
- **现有 IT 环境**（Existing IT Environment）。这个工作产品描述了构成企业现有信息技术概况（landscape）的一些因素，这些因素可以应用于或限制所开发的系统。这个工作产品列出了软件和硬件资源（assets），架构不是可以应用它们就是受限于它们。某个硬件资源可以是组织中的现有的网络或能够访问到的节点。某个软件资源可以是一个已有系统，或许可以用作您的整个解决方案的一部分。后面的“最佳方法：调查现有 IT 环境”部分讨论了如何确定已有 IT 环境的内容。

最佳实践：理解更广义的上下文

这个案例研究假设业务和技术需求是已知的。但是，在某些情况下并非如此。如果不能得到这些输入信息，项目组（尤其是架构师）需要决定如何在没有这些信息的情况下继续，或是否在项目内生成这些项。

最佳实践：调查现有 IT 环境

现在大多数新的解决方案都要考虑棕色地带因素。因此，理解系统面临的 IT 环境十分关键。

IT 架构师和业务分析师很少能像真正的建筑师那样仔细地进行这些分析。不幸的是，在 IT 行业，几乎没有时间花在这些调查上。（Hopkins 2008）

6.3 应用简介

YourTour 是一家将旅行社和其目标顾客连接在一起的公司。YourTour 的特色在于不仅能将多个地点组成一条旅游线路，而且能为所有参加的游客提供所需的住宿和灵活的交通。设想旅行社想开发一条包括 4 个景点的观察野生动物的旅游线路。YourTour 系统不仅能定义详细的行程（和日程），而且可以在住宿和交通方面基于可用的情况下为每个游客提供灵活性（例如，选择四星级酒店而不是二星级酒店，通过飞机而不是火车来旅行）。另一个例子是含有同一大洲或同一国家内的好几个地方的名胜景点观光旅游。

简单来说，YourTour 所提供的服务介于固定线路游（通常是选择单个目的地，有固定的交通和住宿安排）和自助游（为一个人安排这些都很头疼了，更别提一群人了）之间。

在讨论 YourTour 的高层次需求之前，我们解释一下为什么选择这个例子作为研究案例。除了这个案例的概念比较简单，不需要专业知识就能理解这个优点之外，它还给出了许多架构师在真实项目中经常面临的挑战，其中包括：

- **核心功能需求。**YourTour 系统必须满足所有的核心功能需求，例如处理顾客预定以后的支付问题。
- **开发期间质量。**系统必须满足开发期间的质量要求，如可扩展性和可移植性。
- **运行期间质量。**系统必须满足运行期间的质量要求，如性能和可用性。
- **系统约束。**系统的开发必须符合指定的要求，如成本、进度和特定技术的应用。
- **可重用资源。**大多数系统不是重新开发的，而是源于现有的系统或高度重用现有的资源。在整个开发期间，架构师应当考虑已有的可重用资产，如架构风格和模式。
- **系统集成。**YourTour 系统已经采用 IT 技术维护顾客信息，所以，同时也要考虑与现有客户关系管理系统（CRM）的集成。您可以把它看作是可重用的资源，尽管是非常粗粒度的。另外，YourTour 系统还需要与两个第三方系统集成，一个是处理转账的支付引擎，一个用来是查询是否有合适的住宿和车票（并预定）的预定系统。
- **物理分布。**YourTour 系统将部署在公司总部和分公司（顾客可以在那里预定将要出行的旅游团），以及所有支持互联网接入的设备（如浏览器和 PDA）上。系统部署是架构师从既定需求获得适当解决方案的过程中必须面对的另一个复杂问题。

尽管这些考虑并不完全，但它确实能令人体会到架构师所必须处理的问题的复杂性。正如您将在接下来的章节中所见的，一个开发流程能够以系统的方式解决的正是这种复杂性。

YourTour 系统的盈利关键取决于预定的数量（因为 YourTour 按照每个订单收取费用）和运营成本。这两个因素都会受到良好的构架设计、良好设计并良好实现的旅游预约系统的极大影响。如果这个系统能够方便地访问和使用，而且稳定并运行良好，那它就能够吸引更多的旅行社和游客，预定的数量也会相应增长。类似地，如果系统容易安装、管理和维护，运营成本也会更低。

图 6.2 是从 YourTour 系统的**业务实体模型**中抽取出来的。这个图包括了 YourTour 系统支持的重要概念。

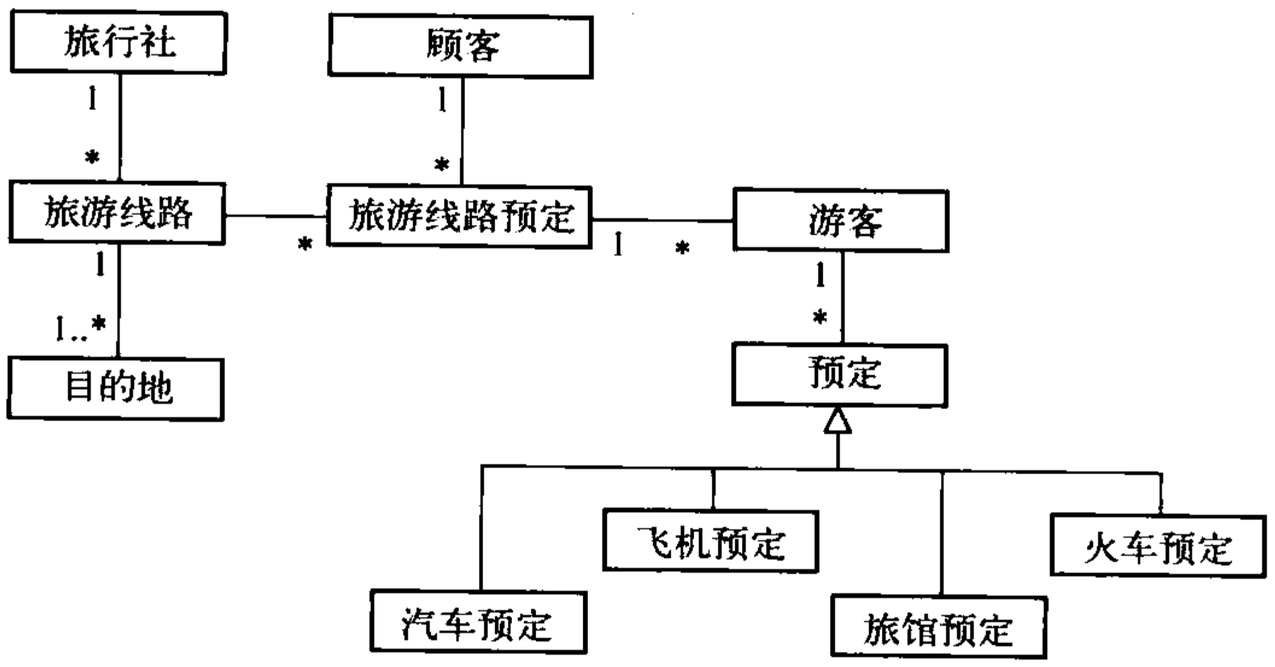


图 6.2 部分业务实体模型

一个旅游线路（tour）由旅行社（tour organizer）预定的几个目的地（tour locations）组成。一个目的地表示游客将访问的地点以及周边地区。YouTour 系统仅仅关注目的地之间的交通问题，而不包括旅游地点附近的安排，例如在同一个目的地的景点之间用私人大巴运送游客。旅行社必须安排这样的运输，还要安排导游、吃饭等。游客（tour anticipant）是预定一次出游的人。

显而易见，旅游线路预定（tour booking）代表了 YourTour 系统的核心动力。它代表游客和一次出游之间的关系。另外，真正的游客和预定的顾客（customer）是有区别的。尽管他们可能是同一个人，顾客是预定并付款的人，而游客是真正参加旅游的人。这种情况类似于一个家长为家庭成员预定了一次出游，可能也包括她自己。每个家庭成员都看作是这次旅行的游客，而付钱的家长是顾客。一次旅游预订间接地包含了与每位游客相关的任何预定（包括住宿和车票）。

6.4 YourTour 的愿景

YourTour 开发团队能够得到的另一个信息是关于 YourTour 的愿景。这个工作产品的内容会在项目图表中（如果存在）找到。愿景可能包含一个问题声明、一组利益相关者、一个特征列表、任何系统必须具备的品质和对系统的任何约束。我们在后面的章节讨论这些内容。

6.4.1 问题声明

YourTour 系统必须解决风险承担者和用户的下列需求：

- 旅行社想得到最广泛的用户群以确保满负荷运行。
- 顾客要求能够访问一个巨大并花样繁多的旅游线路目录，而且能够通过选择住宿和交通的方式来控制每位游客的花费。旅游线路目录列出旅行社提供的所有旅游线路的总价。
- YourTour 的拥有者希望能够为旅行社和客户开展业务提供平台，并希望从每笔订单中抽取利润。
- YourTour 系统必须提供一个安全的环境，从而使操作员可以为旅游线路发布广告、顾客可以预定它们、系统管理员可以有效地提供支持。

6.4.2 利益相关者

利益相关者是指对系统有兴趣或有利害关系的人、团队或组织（或者其中的一部分人）。（IEEE 1471 2000）

表 6.3 简单描述了 YourTour 系统的关键利益相关者。项目开发团队中的各种利益相关者（如**项目经理**或**测试人员**）不进行单独列举，由系统提供商表示。这个列表也不包括住宿和交通服务商，因为他们仅仅和 YourTour 的第三方预订系统交互。

表 6.3 利益相关者概括

角　色	描　述	职　责
系统拥有者	拥有 YourTour 系统的利益相关者	定义、复审 YourTour 系统的关键需求并决定它们的优先级。 定义系统的使用规则，包括旅行社的费用、操作规程等。 能得到系统运营的统计数据，如订单数量和系统的运行状况。 为系统的开发和管理提供资金。
系统提供商	提供 YourTour 系统的利益相关者	理解系统需求并构建满足这些需求的系统。 构建一个符合开发期质量要求（如可维护性和可扩展性）的系统。 采用并遵循一个适当的发布流程。
业务管理员	使用 YourTour 系统执行业务管理职责的利益相关者	收取订单费。 管理系统内容，包括链接（如旅行保险公司）、广告和促销。
顾客	使用 YourTour 系统预定旅游线路的利益相关者	浏览现有的旅游线路。 为一个或多个旅游线路预订游客。 给旅行社付款。
维护人员	系统部署以后负责系统更新的利益相关者	修改交付后的系统，从而再次发布部分或整个系统。
供应商	提供系统开发、运行所需软件、硬件的利益相关者	提供符合规范的软件和硬件。
支持人员	监控和维护系统的利益相关者	为系统的最终用户提供技术支持。
系统管理员	使用 YourTour 进行系统管理的利益相关者	监控系统的活动并收集统计数据。
旅行社	使用 YourTour 系统来发布旅游线路的利益相关者	介绍旅游线路。 定义旅游线路价格，其中不包括住宿和交通费用（这些依赖于游客的选择）。 在每个景点为游客提供方便，如导游、当地的交通和三餐。 为每个订单向系统拥有者付费。

6.4.3 系统功能

YourTour 系统的功能特性需要满足它的利益相关者的需要和他们关心的问题。我们将在第 7 章中详细讲述系统需求。在这里，我们仅提一下 YourTour 系统的高层特性。

正如我们在这一章先前所述，您可能可以获得一个定义 YourTour 应用预期参与的业务活动的业务流程模型。这个模型帮助您找出 YourTour 系统需要的特征。图 6.3 是这个模型的一部分，它显示了如何确定一条旅游线路、登记游客、为每位游客选择预约，最后预定出游。从先前的表 6.3 中列出的利益相关者方面来看，所有这些活动都由顾客进行操作。

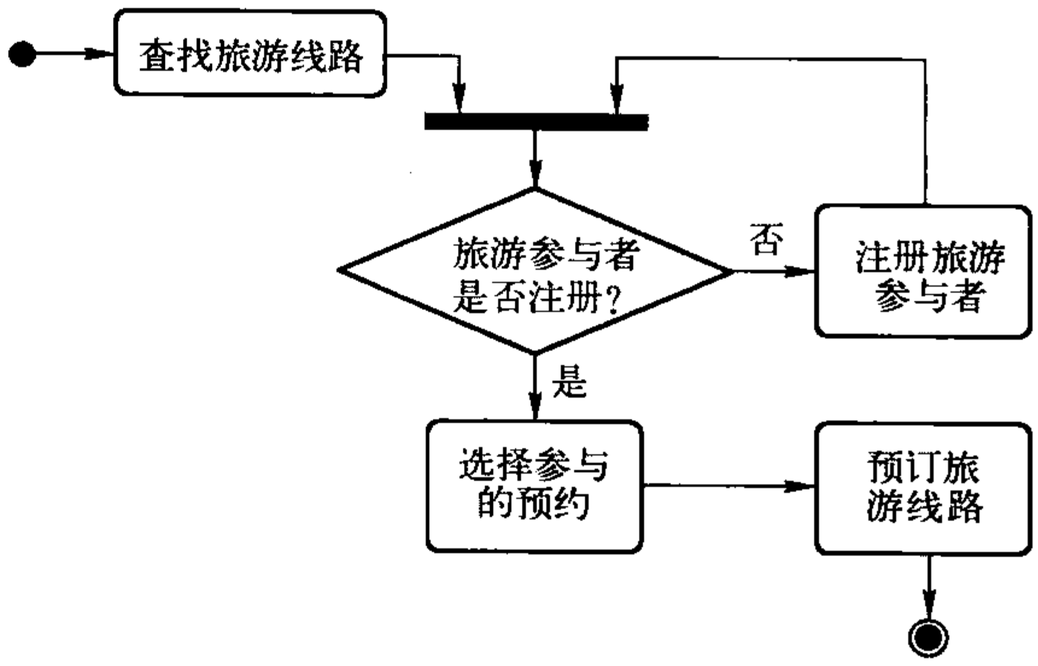


图 6.3 部分业务流程模型

下面是 YourTour 预想的功能列表：

- **设定旅游线路**。旅行社可以设定旅游线路的信息，包括旅游的目的地和价格（住宿和交通由游客选择，并不包括在内）。
- **浏览旅游线路**。顾客可以浏览分类组织起来的旅游线路及详细介绍（包括目的地和价格信息）。系统显示某次出游剩余的名额以及可以得到的优惠。
- **预定旅游线路**。顾客可以预定一次出游。当一个顾客下了订单并付款后，旅行社会收到这笔付款。旅行社为每笔预定向 YourTour 支付一定的费用。
- **管理用户的账号**。系统支持多种类型的用户，如旅行社、顾客或系统管理员，每个用户都有一个账号。系统可以创建账号，如果账户信息不正确，允许修改。系统管理员有管理所有账号的权限，如浏览、更新、备份和删除。
- **管理预定**。旅行社可以取消一条旅游线路（如因为没有导游）或取消一个预定（如顾客的信用记录很糟糕），顾客也可以取消自己的预定（可能会收取定金），管理员可以备份所有预定数据。
- **管理系统安全**。系统拥有安全机制，特别是对可信度高的用户授权，允许他们进行权限范围内的操作。
- **允许访问外部服务**。系统可以访问与旅行有关的外部服务，如保险公司的服务系统。这样 YourTour 可以通过广告和点击率获得利润。

6.4.4 系统的质量

除了功能特性，系统还必须满足一些质量要求和约束。系统的一个重要质量要求是系统的可用性，备份和维护系统时不需要关闭系统。另一个质量要求是性能，特别是系统最终预定确

认的操作（包括预约住宿、交通和付款）不能超过 10 秒钟。可扩展性是另一个令人关心的问题，系统需要支持 5000 个用户同时在线访问。

6.4.5 约束

除了**业务实体模型**和**业务流程模型**，您还需要规定应用必须遵循的**业务规则**。业务规则描述了控制业务如何运行的政策，如“未满 18 岁的人不能预定”。您也可能需要访问一个代表现有 IT 环境的**现有 IT 环境**工作产品。这个工作产品可能包含可以重用的或约束您的因素。也许还存在一个**企业架构规范**工作产品，这个工作产品包含了影响架构构建方式的规则和指导，如一个许可的供应商列表。

从约束的角度来看，YourTour 公司已经决定将和第三方的支付引擎（YourTour 公司已经决定这部分不包括在 YourTour 系统之内）进行交互，还有一个第三方预订系统。公司还决定 YourTour 应用的实现将使用一个现有的遗留系统来存储用户信息。

公司还决定让最终用户能够通过 Internet 用浏览器和移动设备（如 PDA）来访问系统。YourTour 分公司的销售人员可以用浏览器在本地公司局域网透过防火墙访问系统。系统管理员也会以同样的方式管理系统。

6.5 总结

正如我们在这一章的介绍部分所述，这些信息可以在很多工作产品中找到，如**业务实体模型**、**业务流程模型**、**业务规则**、**现有 IT 环境**、**企业架构规范**和**愿景文档**。**愿景文档**特别描述了 YourTour 系统大致所能提供的功能，该文档用作决定项目是否继续进行的主要输入信息。如果项目继续进行，这些信息将作为开发更详细需求的基础，正如第 7 章中所述。

第 7 章

定 义 需 求

这一章将讨论需求规范，尤其关注架构师这个角色。正如第 2 章中所讨论的那样，架构师在需求规范中仅涉及外围的工作，因为**业务分析师**主要负责这些需求相关的任务。然而，我们将用这一整章来讲述这些内容。这是为什么？7.1 节很好地回答了这个问题。

考虑本书的目的，在这一章中，我们假设有好几个工作产品流入或流出**定义需求**这个活动，如图 7.1 所示。我们会在这一章中详细描述这些组成定义需求活动（第 1 章中有所介绍）的任务。

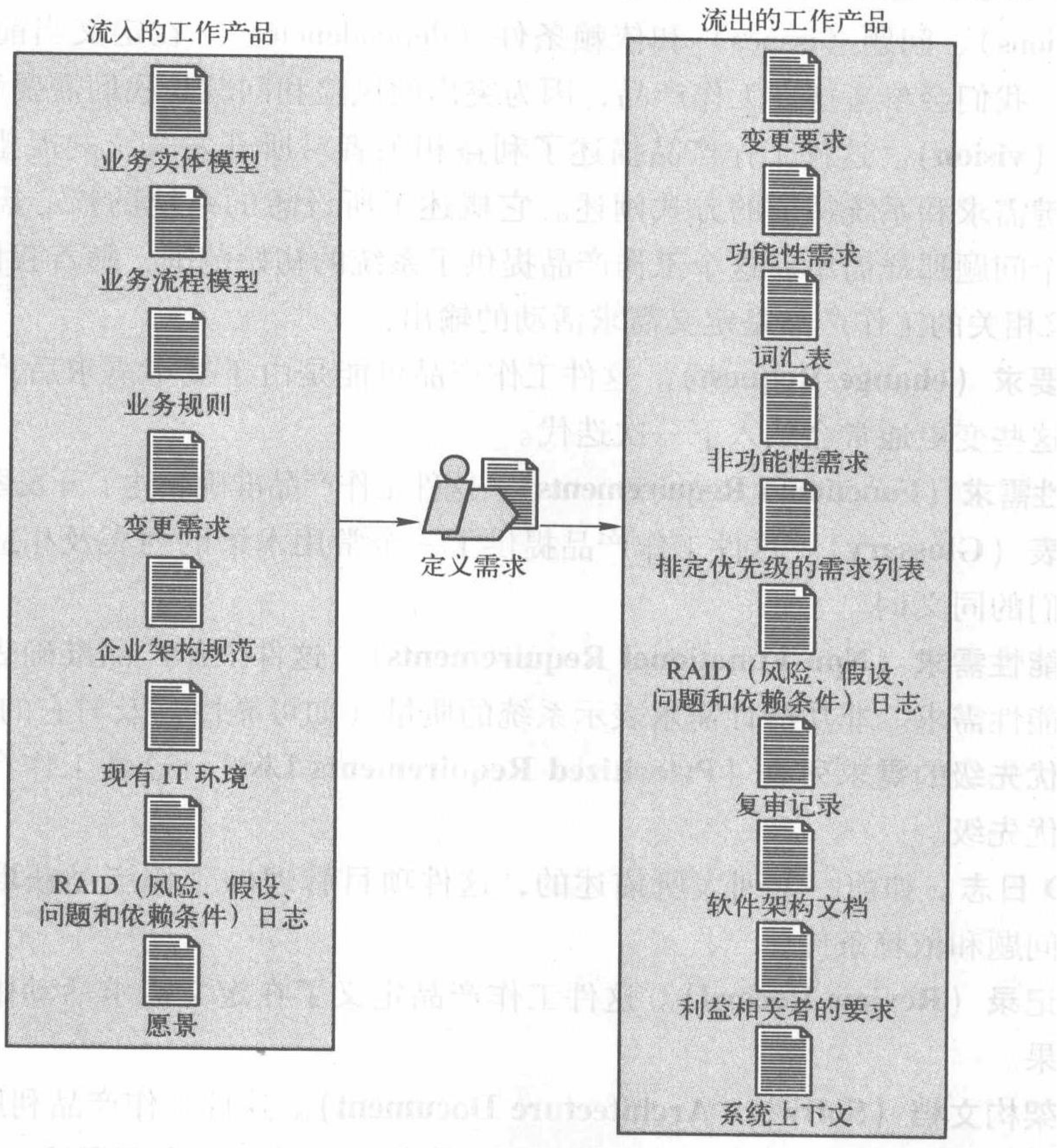


图 7.1　流入和流出定义需求活动的工作产品

我们在第6章中讨论过这些输入的工作产品，如下所示：

- **业务实体模型（Business Entity Model）**。这件工作产品定义了所关注业务领域的关键概念。
- **业务流程模型（Business Process Model）**。这件工作产品定义了业务中需要执行的各种活动和执行这些活动的角色。
- **业务规则（Business Rules）**。这件工作产品定义了所开发系统必须符合的政策或条件。
- **变更要求（Change Request）**。这件工作产品描述了变更系统的需求。这些变更可能很重要并极大地影响当前迭代关注的焦点。
- **企业架构规范（Enterprise Architecture Principles）**。这件工作产品包含了从企业级定义的规则和指导，说明并指导构建架构的方式。
- **现有IT环境（Existing IT Environment）**。这件工作产品准确描述了构成现有IT大环境的因素，这些因素可用于或限制所开发的系统。
- **RAID日志**。这是一件项目管理的工作产品，它收集了项目的风险（risks）、假设（assumptions）、问题（issues）和依赖条件（dependencies）。在定义当前迭代的需求优先级时，我们会参考这件工作产品，因为突出的风险和问题是我们需要考虑的因素。
- **愿景（vision）**。这件工作产品描述了利益相关者对所开发系统的展望，以利益相关者的关键需求和系统特征的方式阐述。它概述了所设想的系统的核心需求，可能就像说明一个问题那样简单。这个工作产品提供了系统的初始范围，随着我们的推进而修正。

以下需求相关的工作产品是定义需求活动的输出：

- **变更要求（change Request）**。这件工作产品可能是由于复审需求后产生的变更而得到的。这些变更通常会纳入下一次迭代。
- **功能性需求（Functional Requirements）**。这件工作产品准确表达了开发系统的功能性需求。
- **术语表（Glossary）**。这件工作产品提供了一个常用术语的列表及相应的解释，还提供了它们的同义词。
- **非功能性需求（Non-Functional Requirements）**。这件工作产品准确表达了开发系统的非功能性需求。非功能性需求表示系统的质量（如可靠性）及对它的限制。
- **排定优先级的需求列表（Prioritized Requirements List）**。这件工作产品说明了每项需求的优先级。
- **RAID日志**。如前一个列表所描述的，这件项目管理的工作产品获取项目的风险、假设、问题和依赖条件。
- **复审记录（Review Record）**。这件工作产品定义了在定义需求活动中对需求进行复审的结果。
- **软件架构文档（Software Architecture Document）**。这件工作产品利用好几个架构的视图来阐述系统的不同方面，从而提供系统架构方面的一个全面概述。
- **利益相关者的要求（Stakeholder Requests）**。这件工作产品包含了所开发系统的利益相

关者可能会提出的要求。可能还包括系统必须遵循的参考标准（如服从监管的标准或组织的标准）。我们会在“任务：收集利益相关者的要求”这一部分讨论要求（request）和需求（requirement）之间的区别。

- **系统上下文（System context）**。这件工作产品将系统描述为一个单一的实体或过程，确认系统和外部实体之间的接口。

系统上下文、功能性需求和非功能性需求这些工作产品描述了系统的需求，作为在客户、最终用户和开发人员之间的一个约定。它们能够使客户和用户确认系统是否会成为他们期望的样子，也能够使开发人员构建出要求的系统。

7.1 关联需求和架构

本书中对需求的强调是受实践中需求和架构之间出现的密切关系所驱动的。尽管架构师仅仅包含在需求规范的外围，但是，他们确实包括在内。因此，理解架构师对需求的帮助以及在什么地方有帮助就十分重要。需求通过以下几种方式和形成的架构发生关系：

- **需求影响架构**。最明显的关系是需求影响解决方案的架构，因为解决方案的元素是按照其是否能够满足规定需求来挑选的（因此，作为架构师，理解您将获得的工作产品是个关键）。当推进架构时，架构师关注那些对架构意义重大的需求，架构意义将在补充内容“概念：架构的意义”中进行讨论。
- **定义良好的需求导致高质量的架构**。由于需求驱动架构的定义，因此，一组定义良好的需求比粗劣定义的需求更能产生高质量的架构。让架构师参与某些特定的需求分析和需求定义的工作，凭借他们的经验，往往会得到质量更好的需求。
- **架构的决定影响需求**。在定义架构时，架构师经常必须在不同的需求之间进行取舍，如平衡性能和成本。结果，业务分析师会得到这些反馈，从而对需求进行必要的调整。架构中某些特定因素的选取也会约束可实现的需求。第三方商业系统的选择，如客户关系管理系统（CRM）应用包，会对需求使用的术语、所提供的功能和获得的质量产生约束。在这种环境中，需求经常会进行精炼以适应所选取的架构元素。需求的任何变更都必须和各位利益相关者协商，当然，这些协商可能需要恰当的正式变更流程。
- **架构定义了子系统的需求**。当架构被分解为各个部分时，需求和架构之间还有另一种不那么明显的关系。实际上，对架构中某一部分的说明已经构成了对那一部分的需求。在开发大型系统时，不同的部分由不同的项目开发，其中的很多部分会外包给第三方，这就尤其有意义。我们会在第 10 章中对这些因素做进一步讨论。

概念：架构的意义

“对架构意义重大”这个短语经常用来形容与架构相关的元素，无论它们是对架构意义重大的需求或对架构意义重大的设计元素等。但是，什么是对架构的意义呢？我们第一次接触这个话题是在第 2 章，但是，我们可以在这里更加具体地说明。一个元素如果出现下列情况，就可以认为对架构意义重大：

- 这个元素与系统的关键功能相关，没有它，您将无法得到可用的系统。
- 这个元素与系统的关键质量相关，例如性能，没有它，您将无法得到可用的系统。
- 这个元素与解决方案的重要约束相关，例如需要和特定的外部系统集成。
- 这个元素能引发特定的技术风险。
- 这个元素带来特定的架构挑战。

7.2 功能性需求和非功能性需求

在继续之前，我们应该解释一下在业内使用的多少有些不一致的词汇——特别是**功能性需求**和**非功能性需求**。词语**功能性需求**的意思相当直白：

> **功能性需求**描述了支持用户目标、任务和活动的（IT）系统的行为（功能或服务）。（Malan 2001）

例如，在一个订单处理系统中，允许客户订购商品是一个清晰的功能性需求。

词语**非功能需求**就不是那么直截了当了，因为从名字看，它描述的是它不是什么而不是它是什么。换句话说，您知道一个非功能性需求不是一个功能性需求。一个简单的定义是：

> **非功能性需求**包括约束和质量。（Malan 2001）

这一定义引出贯穿这一章（和本书）的另外两个术语：**约束**和**质量**。

> **约束**是对我们在提供解决方案时所拥有的自由度的一个限制。（Leffingwell 2000）
>
> （系统）**质量**是利益相关者关心的系统特性和特点，因而影响其对系统的满意度。（Malan 2001）

正如在第2章中所述，您可以从业务（如规章制度和资源约束）和技术（如强制的技术标准和强制的解决方案因素）这两个方面考虑约束。您可以从运行期质量（如性能和可用性）和非运行期质量（如可维护性和可移植性）这两个方面考虑质量。质量和约束是架构师特别感兴趣的东西，因为它们通常代表了架构师工作中最具挑战性的方面，您会在接下来的“任务：概述非功能性需求”中看到这些。请注意，在一些地方除了使用术语**质量**之外还使用了**质量属性**，我们将在补充内容“概念：质量和质量属性”中解释它们。

概念：质量和质量属性

在讨论软件质量时，两个术语经常交替使用：质量和质量属性。以下的摘录概括了它们之间的区别：

> 软件质量是软件拥有所要求的综合属性的程度。（IEEE 1061 1992）

性能、可扩展性和可维护性都是质量属性的例子。然而，本书中倾向于使用术语**质量**，因为它似乎在我们的日常交流中更常用。

功能性需求和非功能性需求通常是相互独立的，结果是，一个系统可能拥有正确的功能却无法满足非功能性需求所定义的质量和约束。因为非功能性需求常常对系统的成功起关键作

用，而且通常对架构师来说极具挑战性，您应该在项目的早期就考虑这些非功能性需求（正如本书建议的）。

7.3 编写需求文档的技术

在编写技术需求和非技术需求的文档时，您需要考虑使用什么技术。尽管您可以用文本描述所有的需求，但也可以使用具有一些可视（visual）建模元素的技术来帮助沟通需求。本书采用的用例驱动开发就是一个例子。这项技术主要用于定义功能性需求和非功能性需求，需要以下先决条件：

- 非功能性需求可能和用例的特定方面相关。例如，一个用例中的某些特殊步骤可能需要执行特定的一段时间。
- 本书中把一些不适合在用例中描述的功能性需求称为系统范围的功能性需求（system-wide functional requirements）（意思是它们不与某一特定的用例相关）。例如，像提供一个安全的系统这样的需求就是一个涉及整个系统的功能性需求，并不依赖某个特定的用例。

在本书中，**功能性需求**工作产品包含所有的功能性需求（既包括通过用例来说明的功能性需求，又包括系统范围的功能性需求）。为了与支持使用模型和模型驱动开发的方式保持一致，我们把用例的集合称为用例模型（use-case model）。《Use Case Modeling》（Bittner 2003）和《Writing Effective Use Cases》（Cockburn 2000）这两本书很好地概述了用例建模。

非功能性需求工作产品包含所有的非功能性需求。它是架构师特别感兴趣的东西，因为它包含许多架构师必须满足的最具挑战性的需求。

7.4 流程应用

正如我们在第 6 章中所述，我们考虑您可以精炼并按您的理解来适应自己流程的各种实践。在这部分，我们进一步研究需求任务，特别是在项目生命周期中它们的典型应用。确切地说，需求任务在整个项目生命周期中会发生变化，如图 7.2 所示。

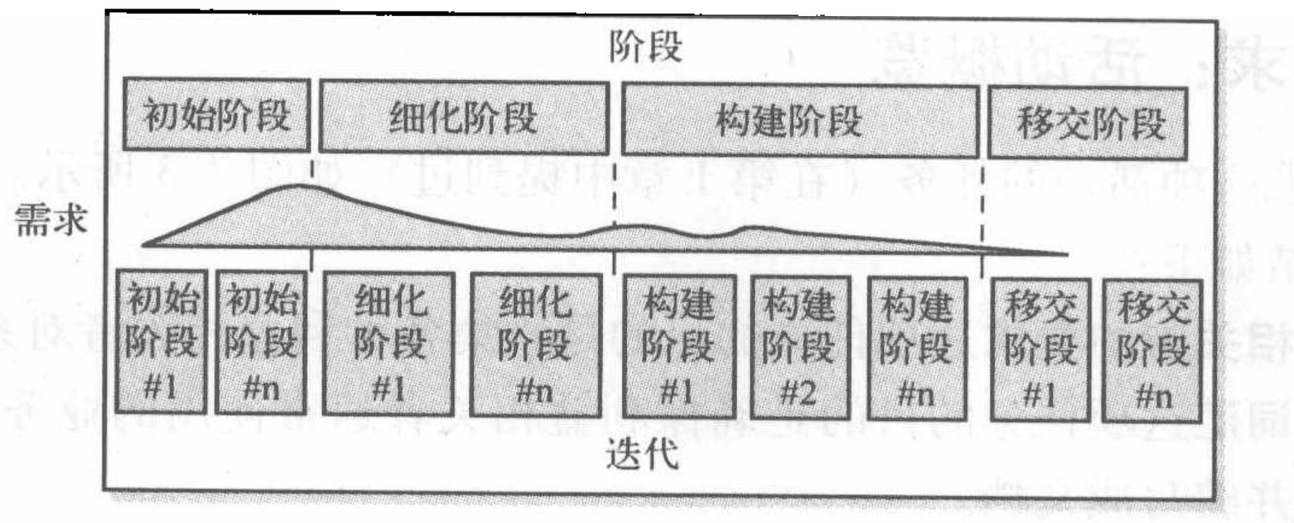


图 7.2 需求和迭代开发

需求定义并不是只在项目开始时发生一次的事情，因为通常不可能事先理解并编写系统的所有需求文档。相反地，对需求的提炼贯穿于整个项目生命周期，这意味着我们需要有效的需

求管理（和变更管理）。需求管理代表一种可以引出、组织和编写系统需求文档以便所有的利益相关者（包括开发人员）理解这些需求的系统方法，它同时也是一种使客户和项目团队对于系统不断变化的需求建立并维持约定的办法。

强调需求活动的最佳时机是在项目初始阶段的后期（此时项目的范围已经确定）和细化阶段的初期（此时对架构意义重大的需求已经被详细描述来推动项目产生一个稳定的架构）。一些需求任务在构建阶段执行，如详细描述剩余需求；有些甚至是在移交阶段执行，如根据系统移交给最终用户后得到的反馈改进需求。

因此，需求随着项目的进行而改进。我们当然不希望大家产生在整个项目过程中需求都有重大修改的印象！但是，当出现如下情况时确实有可能：

- 糟糕的需求管理流程，对项目的影响没有进行任何讨论就接受利益相关者的要求。
- 在详细研究需求时对它产生了更好的领悟。
- 环境出现了一种未预见到的变化，它迫使对需求进行重新研究。例如您的公司和其他组织合并了，现在您必须满足一组新的利益相关者。在这种情况下，暂停项目并回顾前几个阶段以确保已有的决定和假设有效，这可能是有意义的。

7.5 理解任务描述

本书中描述的每项任务在开始时都用一个表格对其进行了概述。表格的内容主要有：

- **目的**部分是这项任务目的的一个简短的摘要。
- **角色**部分列出了执行这项任务的主要角色和次要角色。
- **输入**部分列出了由这项任务引出的工作产品。在迭代式的流程中，工作产品在每次迭代中会不断地改进，每项任务输出的工作产品同时也隐含地看作是这项任务的输入。
- **输出**部分列出了这项任务产生的工作产品。
- **步骤**部分列出了这项任务执行的步骤。
- **架构师角色**部分概述了架构师和这项任务的关系。

7.6 定义需求：活动概览

组成定义需求这项活动的任务（在第1章中提到过）如图7.3所示。

这些任务总结如下：

- **收集利益相关者的要求**这项任务的目的是收集各种利益相关者对系统提出的要求。
- **获取常用词汇**这项任务的目的是确保利益相关者通常使用的业务术语和技术术语被正确地理解并编写成文档。
- **定义系统上下文**这项任务的目的是确保理解所要开发系统的边界范围，同时还识别与系统交互的用户和外部系统。
- **概述功能性需求**这项任务的目的是概要描述一组功能性需求。
- **概述非功能性需求**这项任务的目的是概要描述一组非功能性需求。这些需求可能代表

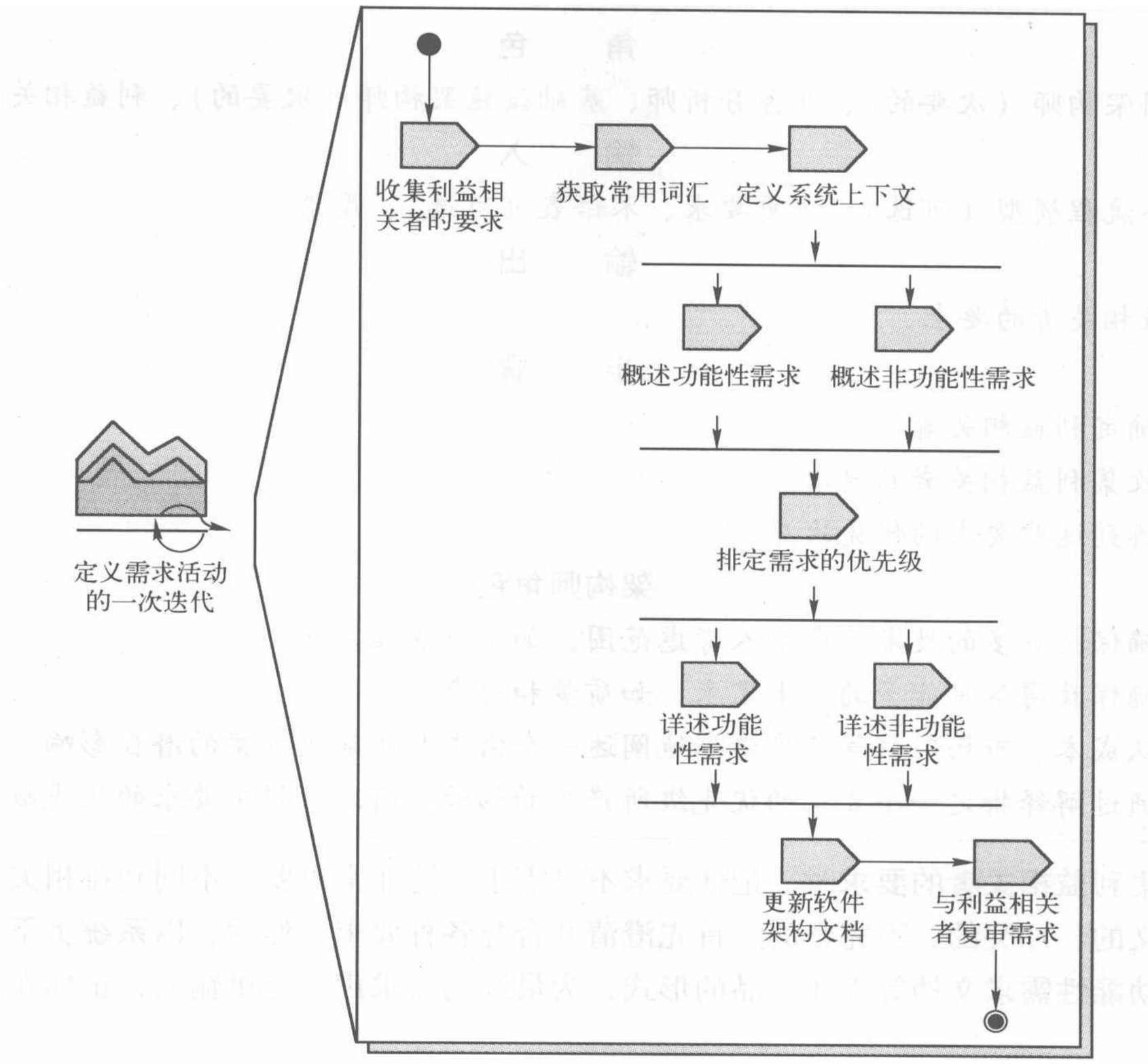


图 7.3 定义需求活动概览

质量或约束。

- **定义需求优先级**这项任务的目的是排定需求的优先级以根据需求的优先级来安排迭代。
- **详述功能性需求**这项任务的目的是定义每个功能性需求以达到从功能层面上推动系统的定义。
- **详述非功能性需求**这项任务的目的是定义每个非功能性需求以达到在系统展现的质量和容纳的约束方面推动系统的定义。
- **更新软件架构文档**这项任务的目的是在软件架构文档中记录对架构意义重大的需求。
- **和利益相关者复审需求**这项任务的目的是使利益相关者认可目前详细程度的需求能够满足他们的要求。

我们在接下来的部分会讨论这些任务，其中，架构师角色会着重描述。

任务：收集利益相关者的要求

这个任务的目的是收集各种利益相关者对系统的要求。

角　色

应用架构师（次要的）、业务分析师、基础设施架构师（次要的）、利益相关者

输　入

业务流程模型（可选）、变更要求、术语表（可选）、愿景

输　出

利益相关者的要求

步　骤

- 确定利益相关者。
- 收集利益相关者的要求。
- 排列这些要求的优先次序。

架构师角色

- 确保把更多的技术角色纳入考虑范围，如系统管理员。
- 确保获得尽可能多的技术需求，如质量和约束。
- 从成本、市场时机等角度清晰地阐述一个需求对于解决方案的潜在影响。
- 通过解释排定一个要求的优先级所产生的影响，有助于排定要求的优先级。

在收集**利益相关者的要求**时，记住要求不是需求，这非常重要。不同利益相关者的要求常常是有歧义的、冲突的、不完整的。首先澄清并合并各种要求，然后，以**系统上下文**、**功能性需求**和**非功能性需求**文档等工作产品的形式，为最终的需求声明提供输入，正如在这一章中稍后所述。

步骤：确定利益相关者

这项任务的第一个步骤是确定哪些利益相关者参与要求收集的活动。通常全部利益相关者都会出现在像**愿景**文档这样的工作产品中，那就是这一项活动的输入。**业务流程模型**也可能会给我们一些关于利益相关者的提示，因为这样的一个模型含有系统将支持的业务活动的描述，因此，也可能包含您需要考虑的代表利益相关者的相关的业务角色。

确保恰当地确定利益相关者，这非常重要，因为不同的利益相关者拥有不同的关注点，在收集他们的要求时，您应该有恰当的关注点并使用恰当的技术。在 YourTour 案例中，利益相关者实际上已经确定了，正如我们在第 6 章中所述。让我们回想一下，他们是：

- 应用拥有者
- 应用提供商
- 业务管理员
- 顾客
- 维护人员
- 供应商
- 支持人员

- 系统管理员
- 旅游组织者

确保不会因为不小心漏掉利益相关者，维护一个利益相关者检查列表通常会很有用，我们会在后面的“检查列表：维护一个利益相关者的检查列表”部分讨论它。

检查列表：维护一个利益相关者的检查列表

您在努力收集需求时，很容易漏掉某个利益相关者。因此，维护并应用一个利益相关者的检查列表是很有用的，在所有软件开发的工作中都应该考虑这一点。

某些利益相关者对架构有特别的意义，因为他们的需求会直接推动架构的某些方面。因此，从一开始就考虑这些利益相关者就很重要。这样的利益相关者包括系统管理员（支持系统的安装、配置和操作）和支持人员（负责提供系统的一线支持）。

步骤：收集利益相关者的要求

这个活动的主要内容是收集利益相关者潜在的需求，因为它们还没有成为约定好的需求，所以称为要求，正如我们在补充内容“缺陷：把要求当作需求”中所述。您可以使用好几种技术达到这一目标，包括使用调查表、开讨论会、头脑风暴会议、访谈等。迭代式开发过程中的另一个要求来源是迭代过程中出现的**变更要求**。这些潜在的需求可能是迭代初期甚至是项目初期（如果是对已有系统的升级）的输入。您应当使用与术语表中定义的术语一致的语言获取这些要求。

当获取系统的功能需求时，这个任务通常执行得较好。当获取那些代表架构师通常最感兴趣的元素的需求时——就是那些本质上属于非功能性的需求（它们代表系统的质量或约束），这个任务常常不能获得一个令人满意的结果。成功的项目在收集利益相关者的要求时就让架构师参与进来，以便从一开始就考虑这些内容。

一个简单有效的技术就是维护这些潜在需求的一个检查列表并确保在收集利益相关者的要求时对这些方面进行讨论。附录 D 提供了这样一个检查列表。这个检查列表可以作为您在收集要求时使用的调查问卷的基础。确切地说，调查问卷的内容将直接用以创建**功能性需求**和**非功能性需求**工作产品，正如我们在这一章中所述。这个方法可以称为架构驱动式的需求启发。

在使用这样的一个调查表时，可能会出现各种各样的缺陷。下面这些“缺陷”的补充内容列出了在收集需求时通常会出现的缺陷。

缺陷：把要求当作需求

要求并不是需求，它是对利益相关者想要系统中有什么的一个表述。在相关各方没有明确同意和与现有需求之间的冲突没有解决之前，它不能成为需求。

当您使用灵活的方法，把客户代表看作是团队的一员时，一个要求就可以是一个需求，可以回避任何冗长的协商。但是，即使在这种情况下，理解要求和需求之间的区别仍然十

分重要，因为一个人可能会提出存在冲突和歧义的要求，在满足这些要求之前必须在它们和现有的需求之间进行调和。

缺陷：购物车式的思维方式

当收集利益相关者的要求时，避免利益相关者购物车式的思维方式，这很重要。在没有任何标准的情况下，要求收集的活动很快就变成了毫无成果的谈话，利益相关者觉得您所提供的任何东西都是绝对需要的。但是，满足这些要求并不是免费的，而且这也不是购物车。您很容易就会掉进这样的陷阱：向利益相关者提供一个类似于购物清单的东西，让他们从中挑挑拣拣。解决这个问题的诀窍是能够用友好的方式向利益相关者指出选择特定需求的后果，如成本、进度和质量，以便让利益相关者理解购买的成本。例如，在问“系统需要是安全的吗”的同时补充“高安全级别的系统有可能会使整个系统的成本翻倍”和对问题不加任何补充说明，在这两种情况下，从同一个人那儿肯定会得到不同的反应。当然，您需要定义什么是安全，但是，请您理解要点。

缺陷：调查问卷过于技术性

可能把给利益相关者的调查问卷当作了技术文档，它的内容超出了利益相关者关心的范围。造成这种情况的原因是很多利益相关者更习惯和更熟悉与业务相关的概念。例如，一个典型的利益相关者理解预定一个游游线路的概念，但是在谈到实用性（availability）时却不熟悉。因此，有时候利益相关者把这种调查问卷看作是项目前进的障碍，而不是获得所建系统关键需求的一个技术。这里的诀窍还是让利益相关者理解花时间回答调查问卷上的问题的价值。给出一个因为没有采用调查问卷而导致出现问题的例子，这通常更容易显示出它的价值！

缺陷：要求过于笼统

另一个缺陷是要求过于笼统。例如，一个要求可能规定任何交易的响应时间少于3 s。然而，在大多数情况下，一些特定交易的响应时间必须遵守这一特定的要求，而其他的交易所花的时间会长得多。如果您不使要求尽可能的明确，这些要求最终会成为需求，系统的元素可能会设计过度，因为这些元素基于不会应用的需求进行设计。

缺陷：要求不可测量

所有的要求都应该是可测量的。“系统将拥有最新的用户界面”，这是一个不可测量要求的例子。这个要求是含糊的，因为“最新”的定义依赖于主观，根本是不可测量的。尽管在搜集要求的过程中仍然允许对各项要求再进行详细的讨论，但是，您必须尽可能确保要求是无歧义的和可测量的。

缺陷：和不适当的人讨论

当把所有的问题都呈现给所有的利益相关者时，有时会出现另一种缺陷。在收集要求的过程中，很多不同类型的利益相关者都有自己的要求，您需要向适当的人问适当的问题。例如，能够对系统的各部分授权是由产品管理提出的要求（可能是在您的组织内部获得的要求），而系统的可用性可能是最终用户提出的（也许是在您的组织外部获得的要求）。

步骤：排定利益相关者要求的优先级

正如先前所述，单个利益相关者对每一个要求都赋予了他们自己的优先级。然而，不同的利益相关者可能提出了同样的要求，但是却给出了不同的优先级。在这个步骤中，您要协调各种要求的优先级，它们最终会影响产生的需求，您和利益相关者会在**和利益相关者复审需求**的任务中复审它们（稍后会在这一章中进行讨论）。

架构师可以出面协调它们，因为他们最有可能从各个方面（如成本、时间）清晰地阐述一个要求的优先级对于实现所带来的影响。

任务：获取常用词汇

目　　的

这个任务的目的是确保利益相关者常用的业务和技术术语能被理解并编写成文档。

角　　色

应用架构师（次要的）、业务分析师、基础设施架构师（次要的）、利益相关者

输　　入

业务实体模型（可选）、业务流程模型（可选）、企业架构规范（可选）、利益相关者的要求、愿景

输　　出

术语表

步　　骤

确认常用术语。

架构师角色

协助定义使用的任何技术术语。

项目应该一贯地使用一个常用术语表，其中的词汇和问题领域的术语一致。确保需求工作产品一贯地使用这些词汇以防项目成员之间产生误解，这尤其重要。这些术语记录在术语表中。每一个术语都应该有一个清晰简洁的定义，而且相关的各个方面也应该同意这些定义。

这个特殊的任务可能会在一个迭代中执行多次并贯穿所有迭代。当需求定义之后，您可以定义和精炼所使用的术语。当需求详细描述之后，您可以再次把它们加入**术语表**。

有好几个原因使得使用一致的术语和架构师密切相关。为了和利益相关者、开发团队有效

地沟通，架构师必须一致、无误地使用这些术语。另外，这些术语经常影响需求和方案元素的命名，因此，适当地定义这些术语很重要。虽然大家通常认为创建术语表是一个相对简单的任务，但是，让人吃惊的是，如此多的组织创建这么简单明了的工作产品时却失败了。这一失败可能会导致后续的歧义和错误表达，从而导致整个项目的拖延，而这些问题从项目一开始就可以避免。

步骤：确定常用术语

术语本身经常来自一个项目的输入工作产品，如**业务实体模型**、**企业架构规范**和项目的**愿景**，或仅仅是描述系统及系统做什么的常用术语，而且它们或许已经包含在**利益相关者要求**这个工作产品的文档中。例如在描述 Yourtour 系统的时候，您可能用到了如旅游线路、旅游组织者、顾客和线路预定等术语。

编写术语的文档通常仅仅包含名称和描述。表 7.1 定义了好几个来自 Yourtour 系统术语表中的术语。架构师经常帮助定义使用的技术术语。特别是在行业标准中已经存在一些技术术语，因此应该直接引用而不是重新定义。架构师还应该仔细地处理同义词和同音异义词，我们在补充内容“最佳实践：协调同义词和同音异义词”中进行了解释。

表 7.1 YourTour 系统术语表

名　称	描　述
顾客	一个顾客代表一个购买旅游线路的人
预定	一次预定代表为一个游客预定的车票和住宿
旅游线路	一个旅游线路代表一次组织到几个地点的旅行
线路预定	线路预定表示游客预定一个指定的旅游线路
旅游地点	一个旅游地点代表一次旅游特定的目的地
旅游组织者	旅游组织者是指销售旅游线路的人或组织
游客	一个游客代表预订了一个旅游线路的人

最佳实践：协调同义词和同音异义词

不同的人很容易用不同的术语描述相同的事物（同义词），也会用相同的术语描述不同的事物（同音异义词），在大型的项目中尤其如此。在构建**术语表**时，确保在所有用例中使用最受欢迎的术语，这很重要（尽管您也许注意到了这些术语的其他名称），因为**术语表**中的这些术语会影响其他工作产品中很多元素的命名。假如说术语保持一致能使我们受益的话，后期对术语名称的修改会导致很多的重复工作。

正如我们之前所述，使用行业常用的术语是值得的。例如，服务（service）这个术语可用于和面向服务架构（SOA，代表通过一个或多个接口提供的一种软件能力）相关的技术上下文中。服务这个术语还可能用于和商业服务管理（BSM，通常代表对支持业务的技术的整合）相关的业务上下文中。

任务：定义系统上下文

目　　标

这个任务的目标是确保理解所开发系统的边界以及确认与该系统交互的最终用户和外部系统。

角　　色

业务分析师、首席架构师（次要的）、利益相关者（次要的）

输　　入

业务实体模型（可选）、业务流程模型（可选）、企业架构规范（可选）、现有 IT 环境（可选）、利益相关者要求、项目目标

输　　出

系统上下文

步　　骤

- 确认参与者（actors）。
- 确认参与者的地点。
- 确认数据流。

架构师角色

- 确保考虑到和系统集成的所有外部系统。
- 确保恰当地定义所有参与者的地点。
- 确保考虑到系统和外部参与者之间的所有数据流。

系统定义的两个关键方面是系统的范围以及系统的边界。在这个任务中，您会确认参与者、他们的地点及参与者和系统之间的数据流，还要把它们编写在所有的**系统上下文**工作产品中。

步骤：确认参与者

在定义系统时，首先需要做的事情之一就是根据系统的参与者及他们与系统的交互方式来确定系统的上下文。参与者是从系统外部和系统交互的某些人或物。参与者可以是一个人、一个外部系统或一个外部设备（如一台打印机）。考虑这些参与者的灵感可能来自于**业务流程模型**工作产品中定义的角色、**企业架构规范**工作产品中的强制性因素、**现有 IT 环境**工作产品中定义的现存因素、**利益相关者要求**工作产品中定义的要求或愿景工作产品中描述的利益相关者列表。

图 7.4 显示了 YourTour 系统中确认的参与者。在这张图中，顾客、销售员、旅游组织者、系统管理员和业务管理员等参与者代表了人（或更确切地说是使用系统的人所扮演的角色），而 CRM 系统、支付引擎和预定系统等参与者代表了外部系统。对利益相关者来说，维护一个参与者列表以确保没有遗漏是有用的，我们会在后面的“检查列表：维护一个参与者列表”部分讨论这些内容。

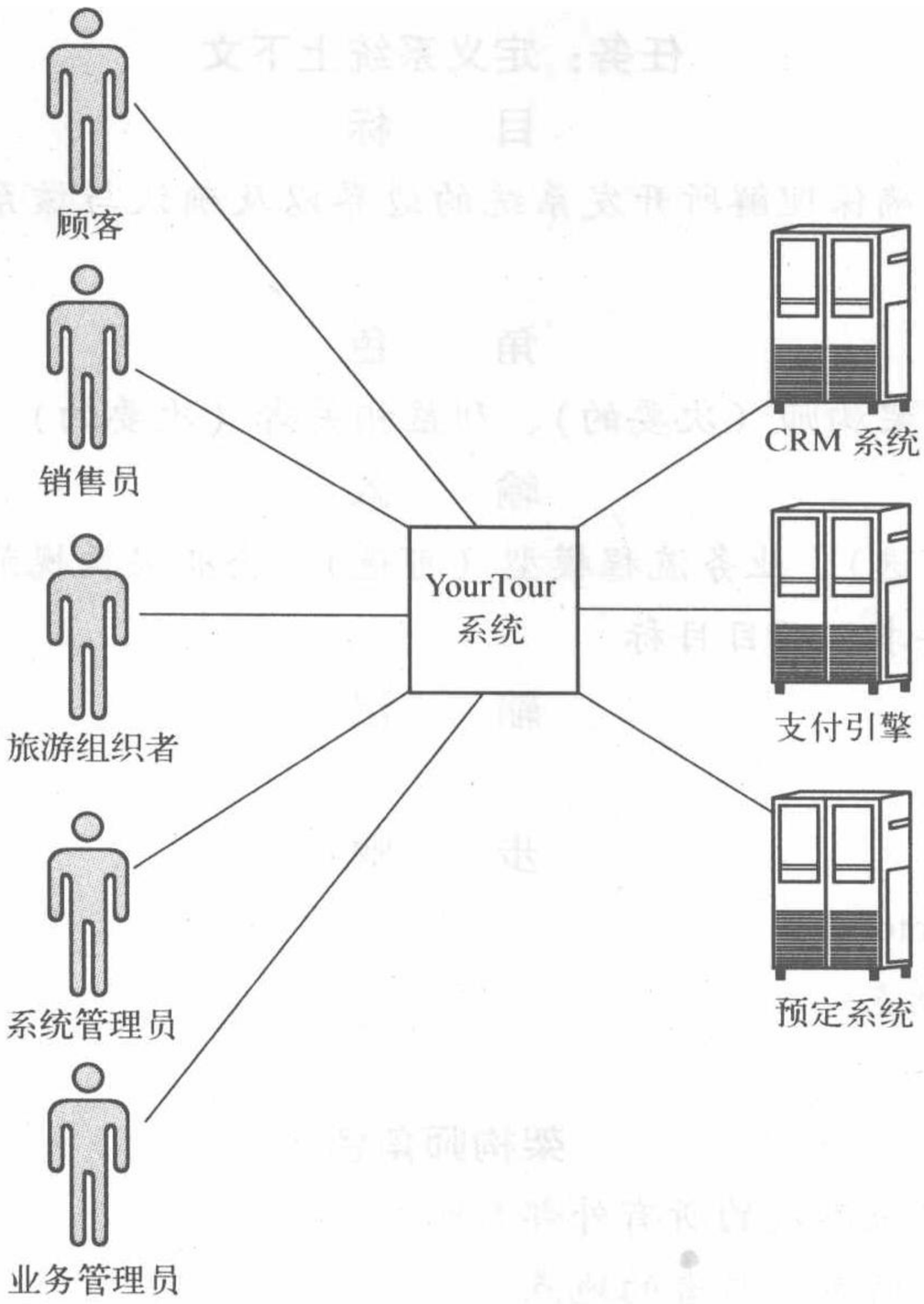


图 7.4 YourTour 系统的系统上下文

检查列表：维护一个参与者列表

正如我们建议维护一个利益相关者列表一样，您可以维护一个参与者列表。作为一项技术，用例建模支持描述与业务相关的功能和业务用户，这些业务用户就由参与者来代表。然而，常常还有另一种参与者与系统交互。除了考虑外部系统和设备之外，考虑确保系统顺利运行的参与者也同样重要，如执行管理职责的系统管理员，还要考虑必要时负责检测的支持操作员。维护和使用这一列表以确保没有漏掉参与者，这很有用。解决这个缺陷可能会导致发现新的利益相关者。

表 7.2 提供了图 7.4 中的每个参与者的简单描述。

表 7.2 参与者的定义

参 与 者	描 述
顾客	顾客是系统的一个用户及旅游线路的潜在购买者
销售员	销售员是系统的一个用户，担当用户或旅游组织者的代理
旅游组织者	旅游组织者是系统的一个用户，他登记现有的旅游线路

（续）

参 与 者	描 述
系统管理员	系统管理员是系统的一个用户，他确保系统顺利地运行
业务管理员	业务管理员是系统的一个用户，他负责系统中与业务相关的信息
CRM 系统	CRM 系统是一个管理与客户相关信息的外部系统
支付引擎	支付引擎是一个用于支持金融账户间转账的外部系统
预定系统	预定系统是一个用于预定与旅游线路有关的住宿和交通的外部系统

如图 7.4 所示，系统上下文关系图不仅有助于理解系统必须面对的外部环境，而且有助于描述系统和外部环境之间预期的接口而又不陷入到系统实现的内部细节中。

参与者和系统之间的交互通常是架构师感兴趣的东西，因为他们关心任何支持这种接口所需的技术和任何同时必须达到的质量（如性能），同时架构师还必须知道可能受到的限制（如接口之间数据的格式）。架构师在弄清楚这些接口的时候常常需要做几个选择并进行适当的权衡。因此，架构师必须确保把代表外部系统的参与者都考虑进来，因为考虑这些系统会导致对特定集成关注点的确认。

步骤：确认参与者的地点

除了确认参与者，知道参与者身在何处也同样重要。参与者的地点随后会影响您的系统的架构，因为您必须知道特定地点所带来的约束。例如，一个分布式环境固有的网络延时会导致系统部件之间的通信速度比非分布式环境中部件之间的通信速度慢几个数量级。表 7.3 描述了 Yourtour 系统的指定位置及与参与者相关的地点。

表 7.3 位置定义

地 点	描 述	参 与 者
远程办公室	在这里，顾客和旅游组织者可以用互联网接入设备（如浏览器、PDA 或手机）访问 YourTour 系统	顾客、旅游组织者
分公司	在这里，顾客和旅游组织者可以接触销售员，预定一个旅游线路	销售员
总公司	在这里，所有远程办公室和分公司的信息被汇总。遗留的 CRM 系统也安置在这里	CRM 系统、系统管理员、业务管理员
Mypay 数据中心	安置 MyPay 支付系统的地点	支付引擎
MyReservation 数据中心	安置 MyReservation 预定系统的地点	MyReservation 预定系统

步骤：确认数据流

您可以更准确地知道系统和外部参与者之间接口的性质，并提供通过这个接口的信息的细节。表 7.4 就是一个例子。架构师可能参与这一活动以确保适当地考虑系统和外部参与者之间的数据流。对数据项的描述通常会参考**业务实体模型**（如果提供了这一工作产品）的描述项。紧接着，确认的数据项例如可以用来描述每个用例中的事件流。

表 7.4 数据流

数据源	目的地	描述	相关数据项
顾客	系统	顾客要求查看所有可预定的旅游线路的信息	旅游线路
顾客	系统	顾客提供游客的信息和预定旅游线路的信息（如付款信息和游客的选择）	游客、游客的选项、预定、付款详细情况
系统	顾客	系统提供可预定旅游线路的详细信息	旅游线路
销售员	系统	销售员查看可预定旅游线路的信息	旅游线路
销售员	系统	销售员提供游客信息和旅游线路预定的信息（如支付信息和游客的选择）	游客、游客的选项、预定、付款详细情况
系统	销售员	系统提供可以预定的旅游线路	旅游线路
旅游组织者	系统	旅游组织者提供旅游线路的描述信息	旅游线路
系统管理员	系统	系统管理员查询用户、旅游线路、游客和预定的信息	用户、旅游线路、游客、预定
业务管理员	系统	业务管理员查询预定的信息； 业务管理员提供旅游费用的信息	预定、游费费用
系统	业务管理员	系统提供相关的预定信息	预定
系统	CRM 系统	系统查询与客户相关的信息； 系统提供与客户相关的信息（创建和更新这些信息时）	客户详细信息
CRM 系统	系统	CRM 系统提供与客户相关的信息	客户详细信息
系统	支付引擎	系统提供付款信息	付款详细信息
支付引擎	系统	支付引擎提供付款确认的信息	支付确认
系统	预定系统	系统提供预定的信息	预定详细信息
预定系统	系统	预定系统提供预定确认的信息	预定确认

任务：概述功能性需求

目　　的

这个任务的目的是概述一组功能性需求。

角　　色

应用架构师（次要的）、业务分析师、利益相关者（次要的）

输　　入

业务流程模型（可选）、企业架构规范（可选）、术语表、利益相关者要求、系统上下文、愿景

输　　出

功能性需求

步　　骤

- 确认功能性需求。
- 概述功能性需求。

架构师角色

确保考虑到更具技术性的参与者（如系统管理员）的需求。

在这个任务中，您会确认并概述系统的功能性需求。我们会基于优先级考虑并详细描述每一个需求，这些优先级会在**定义需求优先级**活动中提供，我们稍后会在这一章中进行讨论。

请注意，确认参与者（在**定义系统上下文**任务中）、功能性需求、非功能性需求并不严格按照先后次序执行。在确认需求时，您可能会发现新的参与者，反之亦然。您也可能获得最好在**非功能性需求**工作产品中获得的需求，我们稍后会在这一章中进行讨论。

步骤：确认功能性需求

在这个步骤中，您将考虑每一个参与者和他们对系统的功能性需求。在本书中，我们使用用例来编写功能性需求文档。在这个步骤中，您还要考虑不适合用用例描述的系统范围的功能性需求。

用例描述的是一个为参与者提供某些价值的完整事件流。找到用例的最好办法是考虑**系统上下文**工作产品中定义的每个参与者需要系统提供的内容，基于已经收集到的**利益相关者要求**，因为系统的存在仅是为了服务于与之交互的参与者的请求。您也可以参考**业务流程模型**和**愿景**工作产品（不仅仅依赖于利益相关者的要求）以确保正在处理的是需要的功能。**企业架构规范**工作产品也会影响功能性需求。例如，“必须符合业界的规定”的规范会要求系统提供运行的详细记录，允许相关的系统参与者访问这些详细记录的用例将支持这一要求。

我们用名称和简要描述的方式来确认每一个用例。用例的名称应当表明通过参与者和用例的交互完成了什么，通常是按照一个动词跟随一个名词的方式（如浏览旅游线路）来命名。用例的简要描述应该反映出用例的目标和动机，这需要参照相关的参与者，同时还要使用**术语表**定义的术语。

一个名称、一个简要描述、有关参与者的确认、用例中主要的数据流和关键的非主要数据流（但不包括驱动数据流的细节）的确认，这些通常是您在流程的这一步骤中所需要的关于用例的全部内容。我们会在这一章后面的**详述功能性需求**任务中补充用例的详细信息时扩展这些信息。

图 7.5 显示了 YourTour 系统初步确定的用例集。架构师通常参与这个活动以确保与较具技术性的参与者（如系统管理员）相关的需求获得确认。

除了这些用例之外，您可能找到系统范围的和用例无法表达的功能性需求。在 YourTour 系统中，我们确定了既需要联机帮助又需要安全。您应该和利益相关者商讨这些需求以得到一个双方都可接受的需求说明，正如我们在补充内容“缺陷：需求是无法协商的”中所述。您还可以从理解这些可能变化的需求中获益，正如我们在补充内容“概念：变更用例”中所述。

缺陷：需求是无法协商的

任何需求都有成本，利益相关者并不总是理解这些成本。架构师经常有利于从资源、时间、技能等方面清晰地阐述这些成本。相对地，利益相关者更有利于从所得到的解决方案角度清晰阐述需求的价值。架构师和利益相关者需要综合这两个方面，从而得到一个双方都可接受和理解的需求。

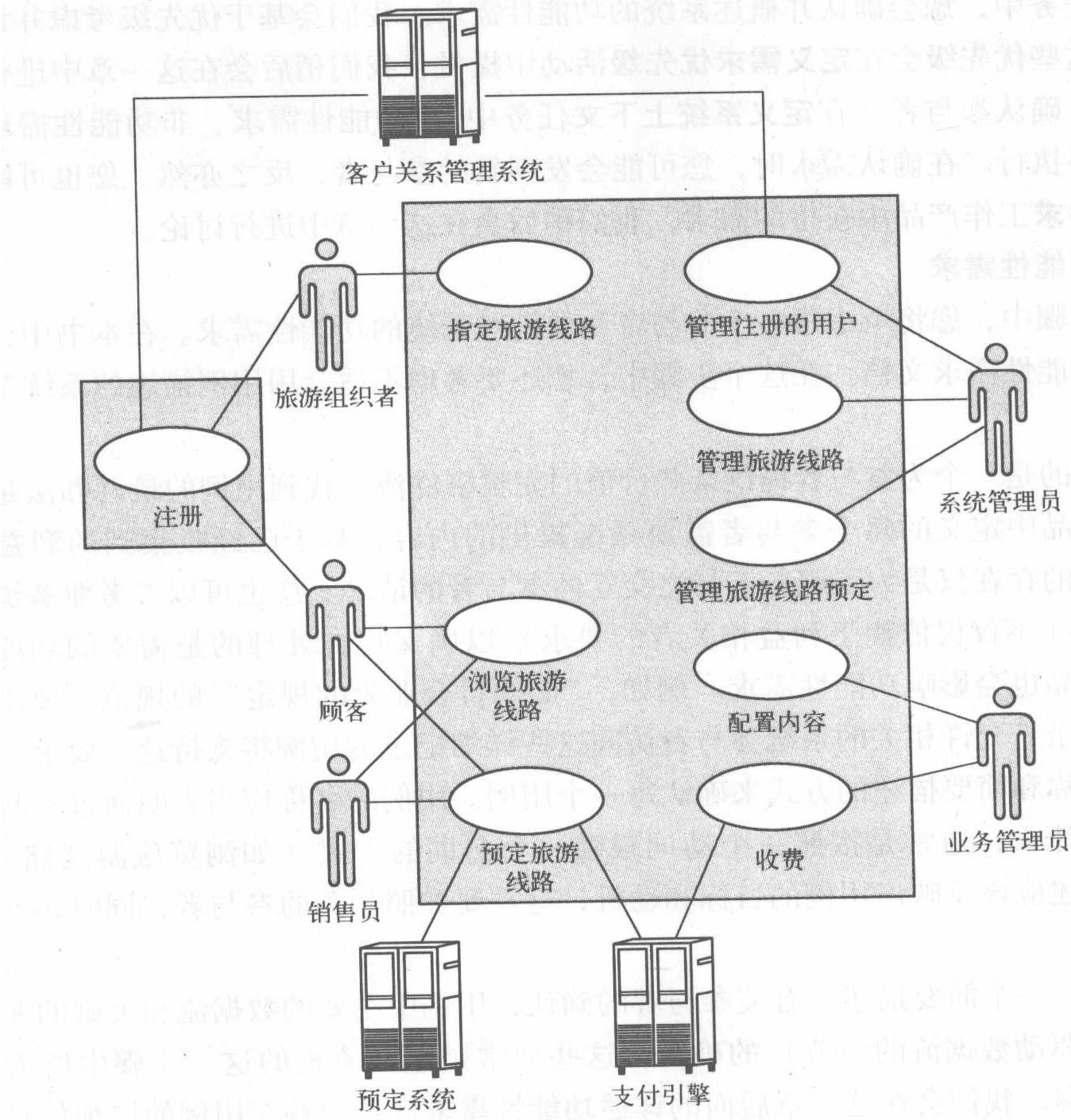


图 7.5 用例模型中的参与者和用例

在一个软件项目中，您几乎能够在任何时间改变任何东西：计划、人员、资金、里程碑、需求、设计、测试。需求可能是软件行业中错误使用最严重的单词，它几乎没有说清楚任何真正需要的东西。几乎所有的事情都需要协商。（Royce 2005）

概念：变更用例

在设计中实现变更用例，以便在变化发生很久之前就做好准备（Ambler 2000）。

一个变更用例代表系统需求的一个潜在改变，它能够让架构师洞察架构的容易改变或扩展的部分。处理变更用例时应当有所权衡，然而根据定义，一个变更用例并不是一个需求，它是一个潜在的需求。尽管没有明确地包含在本书描述的流程中，但是，正式地编写变更需求文档能够证明是有价值的活动。

> 架构师是否应该处理一个变更需求是他需要作出的权衡之一，因为它可能牵涉额外的工作或影响系统的某些质量。假设有两个看似等同的架构选择，一个处理变更用例，而另一个不处理，显然应该选择处理变更用例的架构。理解变更用例会成为需求的可能性有助于架构师进行决策。

步骤：概述功能性需求

在这个步骤中，您要提供功能性需求的一个概述。表7.5描述了YourTour系统已经确认的功能性需求，还指明这些功能性需求是否可以按用例的方式表述或是否是一个系统范围的功能性需求。

表7.5 功能性需求描述

功能性需求	类 型	描 述
注册	用例	注册用例允许一个用户在系统中注册。在旅游组织者指定一个旅游线路之前以及在顾客可以预定一个旅游线路之前，用户必须注册
指定旅游线路	用例	指定旅游线路用例允许一个旅游组织者描述他所提供的旅游线路的各个方面
浏览旅游线路	用例	浏览旅游线路用例允许旅游组织者、顾客和销售员仔细查看现有的旅游线路
预定旅游线路	用例	预定旅游线路用例允许顾客或销售员对旅游线路进行预定，它会引起和支付引擎（用于转账）及预定系统（用于预定特定的住宿和车程）的交互
管理注册用户	用例	管理注册用户用例允许系统管理员管理旅游组织者、顾客和销售员。管理任务包括备份、存档和检索
管理旅游线路	用例	管理旅游线路用例允许系统管理员管理旅游线路
管理旅游线路预定	用例	管理旅游线路预定用例允许系统管理员管理旅游线路的预定
配置内容	用例	配置内容用例允许业务管理员定义可管理的内容，如外部服务的链接（如旅行保险公司）、广告横幅、最新消息和促销
收费	用例	收费用例允许系统管理员收缴关于旅游线路在指定时间内作出的预定的费用，它会引起和支付引擎（用于转账）的交互
联机帮助	系统范围	系统会提供联机帮助
安全	系统范围	系统将提供适当的安全。特别是系统会支持用户认证，还确保用户只能执行获得授权的操作

这些描述给了您一些对用例模型进行重构的提示。特别是我们确认了不同的用户类别：旅游组织者、顾客和销售员。因此，您可能会修正最初的用例模型。图7.6显示了所做的几处修正，包括泛化的参与者和相关的用例，还有确定的管理用例。这些重构通常由业务分析师完成，它还给了架构师关于系统公共行为的提示（尽管您始终必须明确地区分需求和解决方案）。只要重构有助于对**系统上下文**工作产品的理解，那么这些重构就应该在这件工作产品中得到反映。

正如您所见，除了增加好几个用例，我们还加入用户、注册用户、旅游线路预定者等泛化的参与者。这一重构允许您确认通用的行为，还允许您更好地传达用例模型的意图，因为它提供了某种程度的抽象。表7.6给出了新加入的参与者的定义，其他参与者的定义保持不变。新

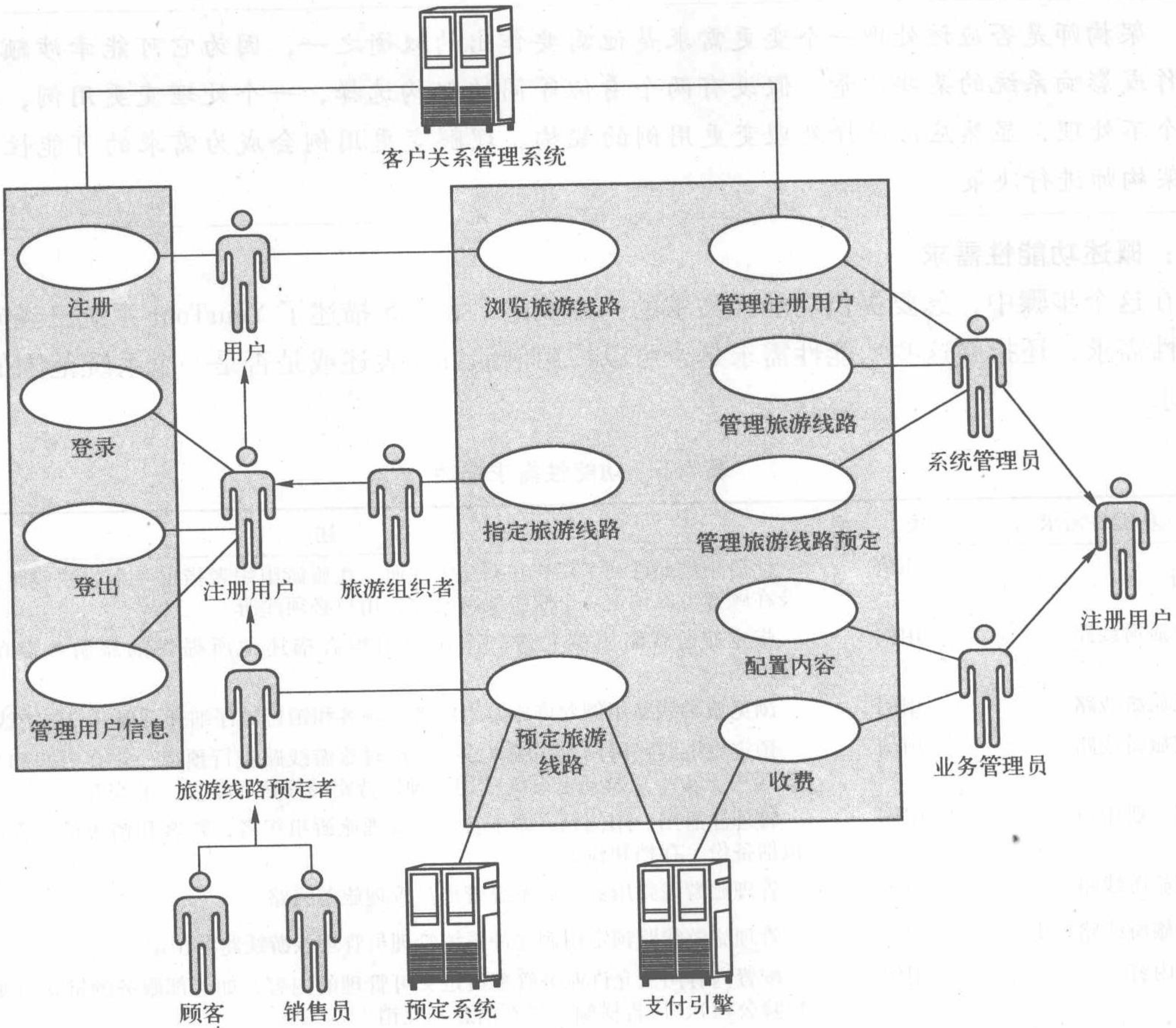


图 7.6 重构的参与者和用例

加的用例一目了然，所以在这里不再详述。

表 7.6 改进的参与者定义

参 与 者	描 述
用户	用户代表 YourTour 系统的一般的和偶尔的使用者，他没有承诺卖出或购买旅游线路。这一角色执行系统最基本的、从这一角色派生出的所有角色都可以执行的一组功能
注册用户	注册用户是指拥有 YourTour 系统账号的用户
旅游线路预定者	旅游线路预定者代表有兴趣购买现有旅游线路的注册用户

尽管没有在图 7.6 中的用例图或相关描述中阐明，但是一些用例的执行存在一定的先决条件。例如旅游组织者不能指定旅游线路，旅游线路预定者只有登入系统后才能预定。另外，还必须遵守一些比较明显的规则。例如，旅游线路预定者不能预定已经满员的旅游线路。这些规则包含在相关的用例描述中（用例描述和某一特定的用例关联），同时还引用**业务规则**工作产

品中的相关项。我们稍后在这一章的**详述功能性需求**活动中讨论这些因素。

任务：概述非功能性需求

目　　的

这一项任务的目的是概述一组非功能性需求。这些需求代表系统的质量或约束。

角　　色

应用架构师（次要的）、业务分析师、基础设施架构师（次要的）、利益相关者（次要的）

输　　入

业务规则（可选）、企业架构规范（可选）、术语表、利益相关者要求、系统上下文、愿景

输　　出

非功能性需求

步　　骤

- 确认非功能性需求。
- 概述非功能性需求。

架构师角色

- 确保考虑到适当的非功能性需求。
- 确保用适当的方式表达非功能性需求。

架构师对确认非功能性需求尤其感兴趣，因为这些需求通常代表架构需要解决的最具挑战性的需求。相对于功能性需求，在这里我们不详细描述非功能性需求。相反地，我们专注于定义它们的意图和目的，可能的话还概述它们的内容。随后我们将根据它在**排定需求优先级**任务中所定义的优先级来详细讲述每一个非功能性需求，我们在这一章后面会讨论这个任务。

值得注意的是，**概述功能性需求**活动和**概述非功能性需求**活动并不按照严格的顺序执行，它们通常同时进行。例如在确认用例时，您可能会遇到和这些用例的特定方面相关的非功能性需求。在这种情况下，可以把这个非功能性需求定义在**非功能性需求**工作产品中并让这个用例引用它。相反地，在确认非功能性需求时，您可能会遇到仅适用于某一特定用例的需求。在这种情况下，这个需求和当前的这个用例而不是整个系统相关。

步骤：确认非功能性需求

主要根据**利益相关者要求**工作产品中指定的要求，您就能够得出非功能性需求。如果您使用**收集利益相关者要求**任务中所讨论的调查表，就很容易理解这个活动，因为这些要求已经按照**非功能性需求**工作产品中的内容分类。这个工作产品总是以相同的类别进行组织，按照可用性（usability）、稳定性（reliability）、性能（performance）和支持能力（supportability）分类，附录 D 中描述了非功能性需求的 FURPS 分类方式。至于利益相关者和参与者，维护一个非功能性需求检查列表是有用的，正如我们在后面的“最佳实践：维护一个系统级的非功能性需求

列表”部分所述。

在确认非功能性需求中，您可以参考**业务规则**工作产品，它明确包含了对您的解决方案的约束，正如我们在后面的“概念：业务规则”部分所述。另外，您可以参考**愿景**文档，因为它可能包含了本质上是非功能性需求的特征，如系统的可靠性。也可以**参考系统上下文**工作产品，因为它定义了和外部参与者的所有交互，它也可能暗示了系统的非功能性需求，如对一个特定外部系统接口的本质的约束。**企业架构规范**工作产品可能包含已定义的技术标准。架构师通常参与这个任务以确保所有相关的非功能性需求都被确认。

代表 YourTour 系统质量的非功能性需求包括易用性（accessibility）、可靠性（availability）、可扩展性（scalability）和可测试性（testability）。代表 YourTour 系统约束的非功能性需求包括进度、集成、平台支持、标准顺应性和第三方组件。

最佳实践：维护一个系统级的非功能性需求列表

和功能性需求不同，一个系统需要考虑的非功能性需求比较有限（尤其是代表系统质量的那些非功能性需求）。无论您用哪种框架对这些非功能性需求进行划分，维护和使用一个适当的非功能性需求检查列表以确保没有遗漏，这都是好的方法。

概念：业务规则

业务规则是定义或约束业务某一方面的声明。其目的是维护业务结构或控制及影响业务行为。（BusinessRules Group 2000）

业务规则描述了一个组织为达到其目标而制定的运营、政策和限制。业务规则可用作和相关第三方进行沟通的机制并使其满足法律和其他法定义务，组织可以借助业务规则达到其商业目标。业务规则还能使组织更有效地运营，因为业务规则可以被自动执行并分析以寻找机会节约时间和成本等。

业务规则分为下面几类：

- 为管理业务行为提供条件的规则（如法律的、监管的、文化的和商业的）
- 为判断一个行动是否成功完成提供标准的规则
- 规定一个行为成功或失败后可以执行什么或不能执行什么的规则
- 指定如何响应对企业有影响的外部事件的规则
- 管理各种业务实体之间关系的规则

步骤：概述非功能性需求

我们在表7.7 中列出了 YourTour 系统**非功能性需求**工作产品中的几个实体。这张表中列出的需求和分类来自附录 D 描述的 FURPS 分类。正如您所料，我们用术语表中的术语来编写非功能性需求文档，架构师通常参与其中以确保正确地阐述确认的非功能性需求。具体地说，非功能性需求必须是可测试的以证明它可以或不可以满足需求。另外，您可以明确地声明什么不是必需的以避免歧义，正如我们在后面的“最佳实践：如果有助于理解，声明什么不是必需

的”部分所述。

表 7.7 YourTour 系统非功能性需求工作产品摘录

分　类	需　求	描　述
可用性（Usability）	易用性（Accessibility）	依照 YourTour 组织的标准，系统将帮助在视力、听力、行动、认知等方面有障碍的人使用
稳定性（Reliability）	可靠性（Availability）	对要求安全的功能（如预定旅游线路），系统在 99.9% 的时间内正常工作。对于其他所有功能（如浏览旅游线路），系统在 99% 的时间内正常工作。系统的备份和维护操作不需要关闭系统
性能	速度	系统会在 10 s 之内处理一个旅游线路预定的确认（包括和 MyPay 支付引擎之间的转账操作和 MyReservation 预定系统之间的预定操作）
支持能力（Supportability）	可扩展性（Scalability）	系统将支持 1 001 000 个注册用户和 5000 个并发用户
支持能力	可测试性	系统将允许记录所有的事务日志，从而可以验证事务及其负载
业务约束	进度	系统将在 6 个月内交付
架构约束	通信	系统将使用行业标准的 Web 协议来实现协同工作
架构约束	遗留系统	YourTour 系统将使用已有的 CRM 系统存储客户信息
开发约束	平台支持	YourTour 系统可以支持 Internet 的设备访问，如 Web 浏览器和 PDA
开发约束	遵循标准	YourTour 系统的开发将遵循 YourTour 开发标准
开发约束	第三方组件	YourTour 系统将不直接处理支付和预定（住宿和交通），它将实现和 Mypay 支付引擎及 MyReservation 预定系统之间的接口

最佳实践：如果有助于理解，声明什么不是必需的

有时候，声明什么不是必需的会有助于防止对需求作出（错误的）的假设和理解不够清晰。一个好的例子就是“YourTour 系统不直接实现处理支付和预定（住宿和车程）的问题，它将分别实现和 Mypay 支付引擎及 MyReservation 预定系统之间的接口”。

任务：定义需求优先级

目　的

这项任务的目的是指定需求的优先级以便可以根据最优的优先级需求安排迭代。

角　色

业务分析师、首席架构师、项目经理（次要的）、利益相关者（次要的）

输　入

术语表、功能性需求、非功能性需求、RAID（风险、假设、问题、依赖条件）日志、利益相关者要求、愿景（可选的）

输　出

排定优先级的需求列表

步　骤

定义需求的优先级

架构师角色

协助定义需求的优先级。

正如在第3章所述，成功地设计一个软件的架构的前提之一是采用迭代式的开发生命周期，它有助于架构师在任何时候都把注意力集中在试图完成的结果上，而不被一件工作产品是否完成所拖累。换句话说，架构师由结果驱动，而不是由文档驱动。更确切地说（正如在第3章中所述），架构师将从关注理解系统的范围和风险转为关注消除风险和稳定架构，再转为关注以递增的方式完成系统。

定义需求的优先级有助于根据您的目标和关注点来决定在当前的迭代中关注什么。因为您的注意力在项目生命周期中是变化的，所以，某一个需求的优先级也会发生变化。因此，定义需求的优先级的任务不会只进行一次。您会在每次迭代中检查相关的优先级，在必要的地方，根据项目当前的状态调整它们（包括新的需求、新的变更要求、新风险的发现和已有风险的消除）。

定义需求的优先级基于好几个因素。和迭代开发方法相关的这些因素之一是确保尽早面对风险。由此，RAID 日志被看作是这个任务的关键输入工作产品。它包括项目已知的现存风险和项目当前需要解决的问题。另一个因素是**愿景**文档中的利益相关者的关键要求和对应的系统特征。愿景文档也是这个活动的关键输入。另外，**利益相关者要求**工作产品也被输入，因为它基于利益相关者最初的要求，暗示了他们的优先级。**术语表**也被输入，因为它包含了所有术语的解释。当然，功能性需求和非功能性需求工作产品也被输入。

每个需求的优先级都记录在**排定优先级的需求列表**工作产品中，它是这个任务的产出。

步骤：定义需求的优先级

通过查看所有的功能性需求和非功能性需求，您能初步定义需求的优先级，从而形成应该在当前迭代中努力处理的高优先级的需求。然而，考虑这些高优先级的需求需要投入相当多的精力，当您考虑每一个需求都可能包含关于详细设计和与其相关的测试的因素时尤其如此。

因此，您需要明智处理这些高优先级的需求。需求并不总是真正独立的这一事实可以在这方面帮助您。例如，如果您解决预定旅游线路的用例，您就懂得它还要解决和第三方组件的集成。为了使处理优先级最高需求的工作量最小化，在指定需求的优先级时，考虑用例和其他需求的一致性是有好处的，下面这个简单的表格就可以说明这一点。

表7.8显示了 YourTour 系统的一个例子。其中，行代表用例，列代表其他需求（表7.5中的系统范围的功能性需求和表7.7中的非功能性需求）。如果一个用例对应一个特定的非功能性需求或系统范围的需求，在相应的表格中就用 Y 表示。然而，您不能通过考虑单个用例来确定系统的进度（必须考虑所有的用例），因此，这张表中没有包含进度。

表 7.8 用例和其他需求的对应关系

	易用性	可靠性	通信	遗留系统集成	联机帮助	平台支持	可扩展性	安全	速度	标准遵循	可测试性	第三方组件
管理注册用户	Y	Y	Y	Y	Y	Y	Y	Y		Y	Y	
管理旅游线路预定	Y	Y	Y		Y	Y	Y	Y		Y	Y	Y
管理旅游线路	Y	Y	Y		Y	Y	Y	Y		Y	Y	
预定旅游线路	Y	Y	Y		Y	Y	Y	Y	Y	Y	Y	Y
浏览旅游线路	Y	Y	Y		Y	Y	Y	Y	Y	Y	Y	
收缴费用	Y	Y	Y		Y	Y	Y	Y	Y	Y	Y	Y
配置内容	Y	Y	Y		Y	Y	Y	Y		Y	Y	
登入	Y	Y	Y		Y	Y	Y	Y		Y	Y	
登出	Y	Y	Y		Y	Y	Y	Y		Y	Y	
管理用户信息	Y	Y	Y	Y	Y	Y	Y	Y		Y	Y	
注册	Y	Y	Y	Y	Y	Y	Y	Y		Y	Y	
指定旅游线路	Y	Y	Y		Y	Y	Y	Y		Y	Y	

这个表有助于报告我们决定的每一个用例的相对优先级。表 7.9 显示了每一个用例排定的优先级及相应的解释。它们的优先级基于表 7.8 中收集的信息和解释中提到的其他特征。每个需求的优先级分为高、中、低。所有解释中提到的风险均来自 **RAID 日志**。**需求优先级列表**既记录了优先级也记录了给出这一优先级的解释。这个列表也可以包含不对应任意特定用例的需求，例如系统必须在指定时间内启动的需求，在这种情况下您尽管可以把它们和对它们的优先级的解释加入列表。您还要确保需求的优先级相互有比较，补充内容“缺陷：所有需求的优先级都相同”会阐述这一点。

表 7.9 排定优先级的用例

优先级	用例	解释
高	预定旅游线路	这个用例代表着 YourTour 系统的核心。它还含有关于响应时间和第三方组件集成的特定需求，两者都被认为具有高风险
高	指定旅游线路	和预定旅游线路的功能一样，指定旅游线路的用例也代表了 YourTour 系统的核心
高	注册	这个用例要求系统和遗留的 CRM 系统集成，它代表了一个特殊的风险
高	收缴费用	这个用例含有关于响应时间和与第三方组件集成的特定需求，两者都被认为具有高风险
中	浏览旅游线路	这个用例比其他用例更需要仔细地考虑用户界面，最好在 YourTour 系统核心功能完成之后再进行考虑
中	登入	这个用例要求考虑认证和授权，这代表一个特定的风险
中	管理旅游线路预定	修改已有的旅游线路预定提出了特殊的挑战，例如能够取消预定并在允许的情况下退款的能力
中	管理内容	这个用例要求考虑内容管理
低	登出	这个用例看作是执行与登入用例相似的操作
低	管理注册用户	这个用例看作是对注册用例的细化
低	管理游程	这一用例看作是对指定旅游线路用例的细化
低	管理用户信息	这一用例看作是对注册用例的细化

缺陷：所有需求的优先级都相同

一个常见的缺陷是所有的需求都被赋予了同样的优先级。无一例外地，有这一缺陷的项目把所有的需求都归为高优先级。如果所有需求含有相同的优先级，那么，为需求定义优先级就是浪费时间，因为这些信息不能用来确认哪些需求相对其他需求更重要。需要定义需求的优先级以便让架构师（或其他人）知道哪些需求对于系统是最重要的。

任务：细化功能性需求

目　　的

这项任务的目的是把功能性需求完善到可以驱动定义系统功能的程度。

角　　色

应用架构师（次要的）、业务分析师、利益相关者（次要的）

输　　入

功能性需求、术语表（可选）、需求优先级列表、利益相关者要求、系统上下文

输　　出

功能性需求

步　　骤

- 细化用例（在当前迭代考虑的每一个用例）。
- 细化用例数据项。
- 细化系统范围的功能性需求。
- 细化功能性需求场景。

架构师角色

- 确保考虑到所有的流（数据流、事件流等）。
- 协助定义用例数据项。
- 协助细化系统范围的功能性需求。
- 协助细化特定场景。

在这时候，您会有一个关于系统需要做什么、系统的范围是什么、需求优先级列表的初步说明。现在您要把注意力转移到改进这个初步的系统定义以获得对需求的更深理解。当前迭代仅仅考虑优先级最高的需求。

在细化各种需求时，添加这些需求的细节可能要求您重新进行已经做过的任务，注意到这点很重要。您可能会发现一个用例需要分成两个或多个用例，或者发现在用例的描述中包含对架构意义重大的细节，因此需要赋予更高的优先级。类似的改进需要纳入到当前或下一次迭代中。

步骤：细化用例

您要考虑在当前的迭代中对每一个用例都执行这一步骤。在这个步骤中，您要从**功能性需求**工作产品开始。它包含一个用例模型，进而包含用例及每个用例的相关描述。您要逐渐地使用例描述更详细，直到利益相关者感到它能清楚地表达他们的需求并满意为止。在细化一个用例时，您要说明以下信息，这些信息已经在**概述功能性需求**任务中进行了定义：

- **名称**。**概述功能性需求**任务初步定义的名称。
- **简要描述**。用例的目标和动机（在**概述功能性需求**任务中已经初步定义）。
- **主要参与者**。初始参与者的名称。如果有两个或多个参与者能够发起这个用例，那就可以定义多个主要参与者。
- **次要参与者**。用例其他相关的参与者的名称。
- **主要事件流**。执行用例时参与者和系统之间交互的主要途径的描述。这些交互已经在**概述功能性需求**任务中初步定义用例时得到简要说明。
- **可选事件流**。当执行用例时可选路径的描述，在参与者和系统之间的交互方面也进行了说明。在**概述功能性需求**任务最初定义用例时，那些关键的可选流（和相关的交互）也已经得到了简要说明。
- **特殊需求**。不在用例事件流中进行考虑、但需要在详细设计和构建时关注的需求的文本描述。这类需求通常代表和用例相关的非功能性需求，如系统质量（性能）或对解决方案的约束。这些内容可能会引用非功能性需求工作产品。本部分稍后会讨论特殊需求。
- **先决条件**。执行这个用例系统所需要的系统状态。
- **后置条件**。用例执行后，系统必须立即所处的可能状态的列表。

在细化用例时，考虑用例的上下文十分重要。用例的上下文以该用例和参与者的关系及该用例和其他用例的关系的形式来描述。如果有价值，可以在一个局部用例图中表示这个上下文，这个用例图只包括这一个用例、它的参与者和与该用例相关的其他用例（如这个用例包含或扩展的用例）。图 7.7 显示了预定旅游线路用例的上下文。

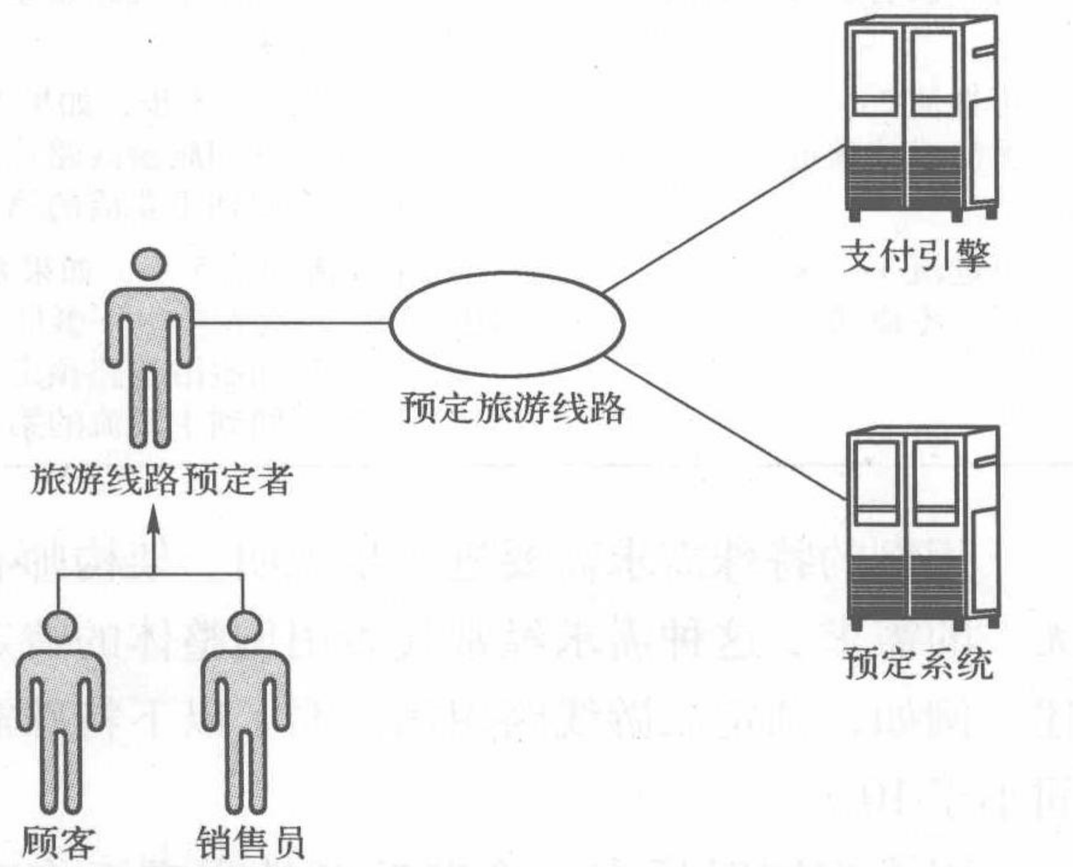


图 7.7 预定旅游线路用例的上下文

用例描述还应该参考**术语表**以确保使用一致的术语。如果需要，您还要参考**利益相关者要求**以更好地理解他们的意图。

还需要进一步解释用例中的一些描述项。我们就从各种事件流开始。一个用例会有几种潜在的事件流通过，如图 7.8 象征性地表示。一个用例总会有一个重要的流（有时候称为基

流或幸福日路径），还可能另外有几个由于例如异常条件等导致的流。

每个事件流的详细描述中应该包括以下信息：

- 用例何时和如何发起
- 用例何时与参与者交互并交换了哪些信息
- 何时应用了业务规则（参考**业务规则**工作产品）
- 用例何时和如何结束

作为一个例子，表7.10中列出了预定旅游线路用例中主要事件流的描述。

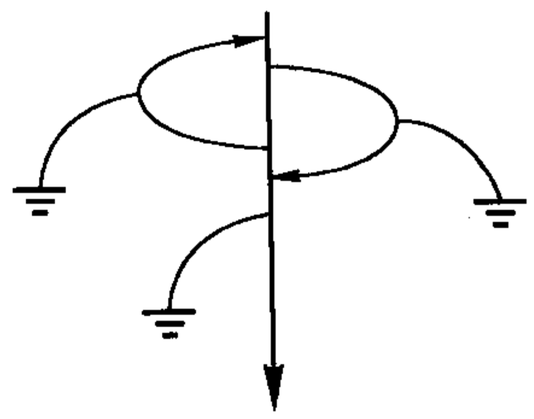


图7.8 一个用例的事件流

表7.10 预定旅游线路用例的主要事件流

名　称	预定旅游线路
主要流	1. 这个用例起始于旅游线路预定者选择为一个或多个游客预定旅游线路 2. 旅游线路预定者确认要预定的旅游线路 3. 旅游线路预定者确认参加这个旅游线路的每位游客 4. 旅游线路预定者为每位游客指定他们想要的住宿和车程选项 5. 旅游线路预定者提供详细的支付信息。预定系统预定住宿和车程，支付引擎处理支付问题 6. 旅游线路预定者收到一个唯一的编号作为旅游线路预定的确认 7. 用例结束

除了详细描述主要事件流，还可能描述可选流。表7.11描述了预定旅游线路的3个可选事件流。架构师可以参与帮助确认可选事件流，因为触发事件流执行的因素本质上可能是技术性的。

表7.11 预定旅游线路用例的可选事件流

名　称	预定旅游线路
可选流1： 游程没有足够的名额	在主要流的第5步，如果没有足够的名额，会发生以下事件： 1. 系统提示旅游线路预定者还有多少个名额 2. 用例结束
可选流2： 无法成功预定	在主要流的第5步，如果无法预定住宿和车程，会发生以下事件： 1. 系统会通知旅游线路预定者无法预定 2. 用例返回到主要流的第5步
可选流3： 付款不成功	在主要流的第5步，如果系统无法处理付款（如支付系统对请求的响应为“不正确的信用卡号”），会发生以下事件： 1. 系统通知旅游线路预定者支付失败的原因 2. 用例返回到主要流的第5步

用例的特殊需求需要进一步说明。架构师有时候会遇到一个和用例相关但和任何事件流都无关的需求。这种需求经常代表用例整体的特定质量或约束，在用例描述中作为特殊需求来描述。例如，预定旅游线路用例声明了以下特殊需求：从提交预定到用户得到预定确认的响应时间小于10 s。

因为您同时还有一个**非功能性需求**工作产品，这就带来一个问题——这些特定用例的和本质上属于非功能性的需求应该放入**非功能性需求**工作产品的哪个物理文件中。一方

面，在用例描述的上下文中看它显然很有价值。另一方面，所有的非功能性需求都放在一个地方（无论这些需求是否和一个用例相关），这样做也是有价值的，对于架构师来说尤其如此。

在理想情况中，会有合适的开发工具使您能够在非功能需求列表中看到特定用例的非功能性需求和用例描述。在现实中，我们经常把这些需求和用例放在一起，然后在单独的文档、电子表格或提供全部非功能性需求概要的工具中保存对这些需求的引用。

用例描述中其他值得讨论的项是先决条件和后置条件。在处理用例时，在用例执行之前，您有时需要确定系统处于某种特定状态。一个普通的例子是“用户已经登入”状态。表 7.12 显示了预定旅游线路用例指定的先决条件。

表 7.12 预定旅游线路用例的先决条件

先决条件	描述
旅游线路预定者已经登入	在预定一个旅游线路之前，旅游线路预定者必须登入系统
旅游线路预定者已经指定一个旅游线路	旅游线路预定者已经指定了他们感兴趣的旅游线路

描述在用例完成后必须确定什么内容，这也是必要的。表 7.13 显示了预定旅游线路用例的后置条件。

表 7.13 预定旅游线路用例的后置条件

后置条件	描述
旅游线路的名额数量相应减少	旅游线路预定后，按照预定的数量减少旅游线路名额的数量

步骤：细化用例数据项

在**定义系统上下文**任务中，您确认了外部参与者和系统之间流动的数据，也确认了这两者在交互中涉及的数据项。然后，您在描述需求的各个方面时使用了这些数据项。这个步骤的目的是进一步解释那些确认的数据项。正如在第 8 章中所述，在考虑需求的解决方案时，您会进一步考虑这些数据项。

我们还是使用预定旅游线路作为例子。简单地看一下主要的事件流，您会发现这个用例用到了游客、游客选项、旅游线路、旅游地点、支付详细信息和旅游线路预定等数据项，它们在**定义系统上下文**任务中就得到了初步的确认。

例如，用例中的“旅游线路预定者收到一个唯一编号，作为预定的确认”这一步告诉您两件事情：

- 可能存在一个预定确认数据项。
- 一个旅游线路预定有一个唯一的引用。

如果您仔细地查看所有的用例，就会看到最终各种数据项的定义被精炼了。只要需要，所

有确认的数据项都会加入到**系统上下文**工作产品中。

步骤：细化系统范围的功能性需求

对于不容易在用例中表达的功能性需求，您通常会修饰这些已经概述的需求的文本描述。例如您可以通过详细描述提供联机帮助的形式来精炼像“系统应该提供联机帮助”这样的需求。您或许会加上“在任何时候都能访问联机帮助，能够以 PDF 文件格式下载帮助的内容。”架构师可以参与来帮助完成这步工作，因为任何系统范围的功能都可能对架构有重要影响。

步骤：细化功能性需求场景

截至目前讨论的功能性需求通常能够覆盖所有可能出现的场景。定义几个具体的场景来示例和阐明某个特定的功能性需求也是有益的。这种方法也能够确保系统必须提供的某一特定价值得以清晰地展示，还能够为开发人员和测试人员提供有价值的信息。

一个和预定旅游线路相关的特定场景可能是类似于这样的一个陈述：“约翰·史密斯是一个现有的客户。他要为三个儿子——汤姆、迪克和哈里预定环游非洲的旅游线路。”

即使是这些简单的陈述，也能使那些本来或许没被注意到的各项变得清晰起来。例如，一个现有顾客的概念暗示系统包含顾客信息。另外，这个特定的场景也不需要顾客参与旅游，因此，顾客为其他人预定旅游线路是非常有可能的。

任务：细化非功能性需求

目　　的

这项任务的目的是把非功能性需求完善到可以驱动系统在展现质量和适应约束方面的定义的程度。

角　　色

应用架构师（次要的）、业务分析师、基础设施架构师（次要的）、利益相关者（次要的）

输　　入

术语表（可选）、非功能性需求、需求优先级列表、利益相关者要求

输　　出

非功能性需求

步　　骤

- 细化非功能性需求（当前迭代考虑的各项非功能性需求）。
- 细化非功能性需求场景。

架构师角色

- 确保以恰当的方式描述所有的非功能性需求。
- 协助处理特定的场景。

在这个任务中，您要细化当前迭代中与高优先级项有关的那些非功能性需求。就像对于功能性需求一样，您可能遇到没有阐述过的非功能性需求或某些需要在非功能性需求中进行重要调整的特性。像这样的精炼可以再次纳入当前的迭代或下一次的迭代。

步骤：细化非功能性需求

这个步骤的目的是讨论相关的非功能性需求及任何需要的额外细节。假设您有这样一个简单的需求“系统将是可扩展的”。除非您能从用户数量、数据量等方面理解它的意思，否则您无法处理这个需求。例如，支持 100 000 个并发用户的方案与支持 100 个并发用户的方案相比，两者完全可能不同。需求的架构方案可能由于需求的细节而发生变化。另一个例子是“系统将实现与 ABC 外部系统的接口”。您必须用关于 ABC 系统接口的详细信息来细化这个需求，以便我们的系统能够以恰当的方式和 ABC 系统通信。

补充每个非功能性需求的一个简单方法是应用 SMART 缩写中的各个元素。这个缩写有许多变体，但是根据我们讨论的目的，假定这些字母代表 specific（明确的）、measurable（可测量的）、achievable（可完成的）、realistic（现实的）和 time - based（基于时间的）。仔细考虑一下这个缩写的 5 个元素并把它们应用到每个非功能性需求的定义中，这样做能够帮助您提供所需的细节。特别地，如果一个需求无法测量，它就无法得到验证。如果一个需求不能得到验证，解决方案中就无法实现它，因为没有办法判断这个需求是否获得了满足。通常需要花很多精力才能提供足够的描述。正如《Principles of Software Engineering Management》（Gilb 1998）中言简意赅地说到：

> 质量属性总是能够做到可测量。这常常看起来很难，但总有方法做到这一点。
>
> (Gilb 1998)

每个非功能性需求的描述也应该参考**术语表**以确保使用一致的术语。如果需要，您还可以参考**利益相关者要求**以更好地理解需求的意图。

步骤：细化非功能性需求场景

细化与非功能性需求相关的场景主要关注质量。《Software Architecture in Practice，2nd ed.》（Bass 2003）提供了一个细化这些场景的技术。在本书中，每个质量都通过一个或多个质量属性场景来进行证明，这些质量属性场景在《Quality Attribute Workshop》（QAW）（Barbacci 2003）中得到确认。每个场景由 6 个部分组成：

- **触发**。触发定义了必须由系统处理的情况。
- **触发源**。触发源定义了导致出现这个场景的角色、系统或系统元素。
- **触发对象**。这个对象代表触发的目标，可能是整个系统或系统的一部分。
- **环境**。环境定义了触发发生时的情况。
- **响应**。响应代表系统在环境所描述的情况下系统对于触发所作出的响应。
- **响应度量**。响应度量是如何处理结果的定义。

我们可以通过考虑一个可扩展性场景来举例说明：

- **触发**。开始一个旅游线路预定。

- **触发源**。使用预定功能的 YourTour 的最终用户。
- **触发对象**。YourTour 系统。
- **环境**。有 5000 个用户同时登录系统。
- **响应**。由于达到了极限，预定请求被拒绝（假设系统定义了最多支持 5000 个并发用户）。
- **响应度量**。启动旅游线路预定的最终用户被告知无法预定旅游线路并被告知应该做什么。

以这种方式描述可测量性场景比简单地声明“系统必须能够支持 5000 个并发用户”更清晰、更可测量，因为它描述了场景发生的条件（源、触发和环境）、场景应用于什么（触发对象）和预期的结果（响应和响应度量）。

任务：更新软件架构文档

目　　的

这项任务的目的是把对于架构有重要意义的需求编写在软件架构文档中。

角　　色

首席架构师

输　　入

业务流程模型、功能性需求、术语表、非功能性需求、RAID 日志、利益相关者要求、系统上下文

输　　出

软件架构文档

步　　骤

更新软件架构文档。

架构师角色

架构师负责这个任务

软件架构文档中的需求观点用于编写对于架构有重要意义的需求的文档。正如在第 4 章“编写软件架构文档”中所述，好几种交叉视图允许您考虑架构的一个特定关注点，而这个关注点又允许您描述需求的一组特定的关注。附录 B 列出了和每个交叉视图相关的需求的检查列表。

步骤：更新软件架构文档

在这个步骤中，您会更新**软件架构文档**中的需求视图和任何交叉视图。这些需求视图包含了对架构有重要影响的功能性需求和非功能性需求。您应该始终牢记：**软件架构文档**的目的是用来沟通架构的。从需求的角度来看，您正在设法沟通对架构有重要影响的需求并解释架构为什么是现在的这个样子。例如，当您在复审架构的过程中必须为您的架构辩护的时候，这些信息特别有价值。

任务：和利益相关者复审需求

目　　的

这项任务的目的是使利益相关者同意需求达到当前的详细程度已经能够满足他们的要求。

角　　色

应用架构师（次要的）、业务分析师、数据架构师（次要的）、基础设施架构师（次要的）、首席架构师、利益相关者

输　　入

业务实体模型、业务流程模型、业务规则、企业架构规范、现有 IT 环境、功能需求、术语表、非功能性需求、排定优先级的需求列表、RAID 日志、软件架构文档、利益相关者要求、系统上下文、项目愿景

输　　出

变更要求、RAID 日志、复审记录

步　　骤

- 定义工作产品的基线。
- 汇集工作产品。
- 复审工作产品。

架构师角色

- 参与复审。
- 回答任何技术问题。

和利益相关者一起复审需求的目的是确保需求达到目前的详细程度已经能够满足他们的要求。在复审各种需求之前，您要定义工作产品的基线以便清晰地记录复审的内容，在必要时还能够参考这些工作产品（例如，为了澄清其中一个工作产品的复审意见）。

步骤：定义工作产品的基线

定义基线是配置管理（它还包括工作产品的版本控制）的一个方面，配置管理的方式是应该在项目的最初阶段就进行考虑并解决的关注点。在这个步骤中，各种需求工作产品会被制定基线，这仅仅意味着将要进行复审的工作产品的正确版本得到了清晰的确认。

虽然一个配置管理方法可能非常简单（例如当手工修改文件的名称以便包括一个版本号时，用电子表格描述工作产品的基线），但是，您还是会发现，即使对于小规模团队来说，利用适当的工具来支持配置管理也是一个效率高很多而错误少很多的方法。

步骤：汇集工作产品

在这个步骤中，您要按照利益相关者能够进行复审的形式来汇集在当前迭代中创建的或更新的（和定义了基线的）需求工作产品。工作产品可能按各种各样的形式进行定义（例如作为一个建模工具或在一个共享工作区中独立的文档）并必须汇集为一个或多个可交付件。汇集

的这些工作产品有可能包括在**定义需求**活动中作为输入或输出的所有工作产品。

步骤：复审工作产品

在这个步骤中，您需要复审这些需求工作产品，还要把工作产品中发现的任何问题都记录在**变更要求**中。记录 **RAID 日志**中出现的任何问题。当复审结束时，您要在**复审记录**中简要地记录复审结果，包括所有的行动项。

复审作为迭代的质量把关以确保这些需求工作产品足够成熟，能支持后续的任务。要使开发团队以外的人（如用户和客户）能够接受当前阶段的工作产品，这无疑很重要，尽管所有的利益相关者都多多少少参与了这一章讨论的多项任务。他们应该检查这些工作产品是否准确地记录了详细的需求。使开发团队的成员参与进来以确保他们理解需要构建的是什么，还要确认哪些地方需要更多的信息，这同样也很重要。

7.7 总结

在这一章中我们讲述了编写系统需求文档的过程，产生下列关键工作产品：**术语表**、**利益相关者要求**、**系统上下文**、**功能性需求**、**非功能性需求**和**排定优先级的需求列表**。我们还根据对架构有重要影响的需求更新**软件架构文档**。这些工作产品驱动了第 8 章和第 9 章中的开发活动。

第 8 章

创建逻辑架构

本章将讨论创建一个逻辑架构的任务，这个逻辑架构实现第 7 章中所定义的需求。补充内容“概念：逻辑架构与物理架构的比较”中阐述了逻辑架构的含义和它如何区别于物理架构。图 8.1 显示了流入和流出**创建逻辑架构**活动的工作产品，这个活动包含了我们在这一章中讨论

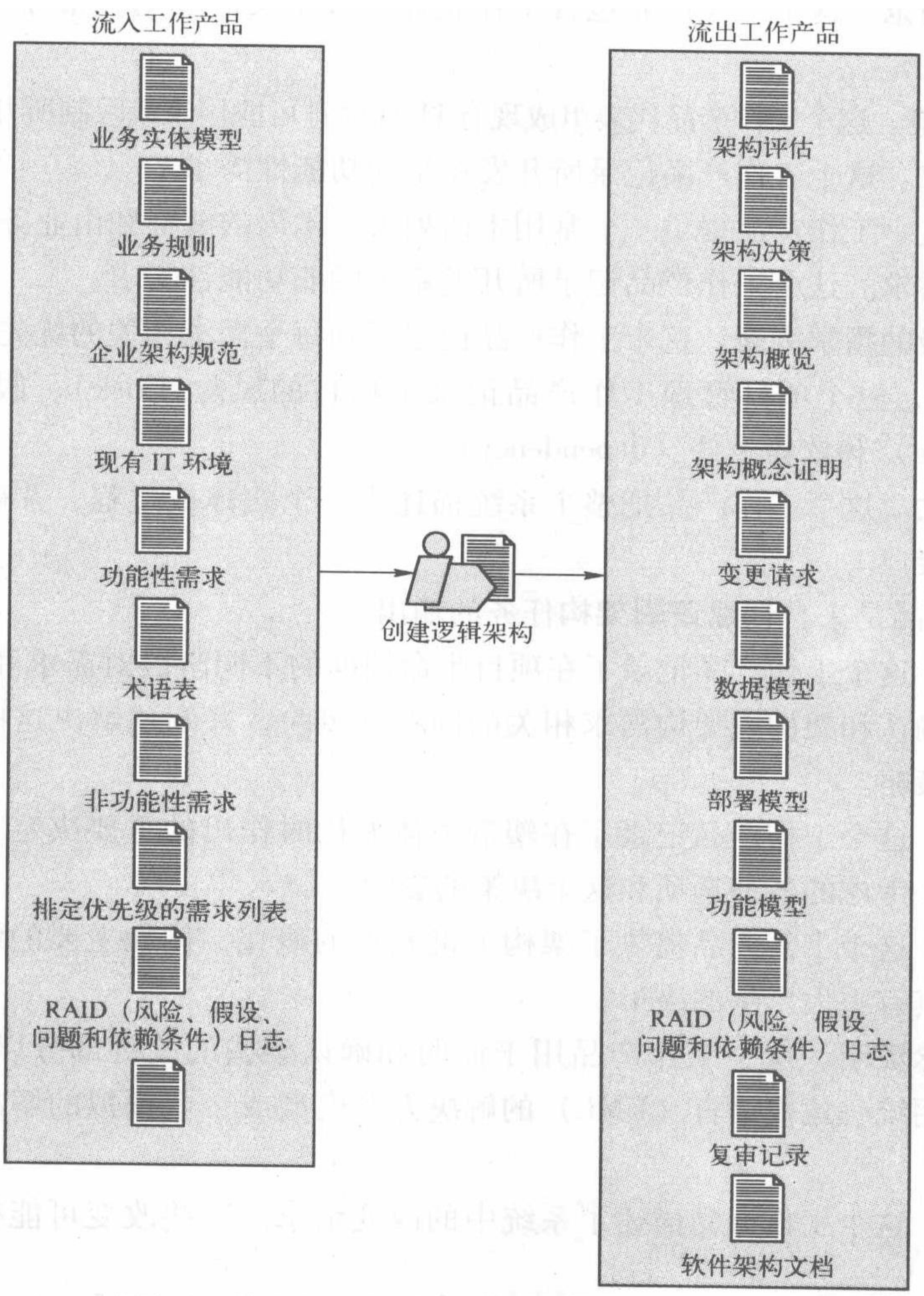


图 8.1　流入和流出创建逻辑架构活动的工作产品

的所有任务。

概念：逻辑架构与物理架构的比较

逻辑架构代表从需求通往解决方案的第一步——第一次主要从独立于技术的角度考虑架构。另一方面，物理架构则更为具体——需要考虑技术。例如，逻辑架构可能包含一个账户（Account）组件，而物理架构可能使用 Java 企业版（Java EE）并把这个逻辑组件实现为一个 Account EJB（Enterprise JavaBean）。

流入的工作产品（在第7章中讨论的）主要有：

- **业务实体模型**。这个工作产品定义了考虑中的业务领域中的关键概念。
- **业务规则**。这个工作产品定义了开发中的系统必须满足的政策或条件。
- **企业架构规范**。这个工作产品包含了在企业层级定义的规则和指导（通知和引导架构的创建方式）。
- **现有 IT 环境**。这个工作产品代表组成现有 IT 环境且可能用于或限制所开发系统的元素集。
- **功能性需求**。这个工作产品记录所开发系统的功能性需求。
- **术语表**。这个工作产品提供一个常用术语列表、术语的解释和由业务使用的可选术语。
- **非功能性需求**。这个工作产品记录所开发系统的非功能性需求。
- **排定优先级的需求列表**。这个工作产品记录了和每个需求相关的优先级。
- **RAID 日志**。这个项目管理工作产品记录了项目的风险（risk）、假设（assumption）、问题（issue）和依赖条件（dependency）。
- **系统上下文**。这个工作产品把整个系统描述为一个实体或过程，并确定系统和外部实体之间的接口。

下面的工作产品是来自**创建逻辑架构**任务的输出：

- **架构评估**。这个工作产品记录了在项目生命周期的不同阶段对需求和架构的评估结果。这些评估确认和架构或架构需求相关的问题和风险，并阐述解决这些问题和风险的行动和缓解策略。
- **架构决策**。这个工作产品记录了在塑造整体架构时作出的重要决定。它阐述了这些决策的理由、考虑的各种选项和这个决策的影响。
- **架构概览**。这个工作产品提供了架构关键元素的概述，例如主要的结构性元素、重要的行为和具有重大影响的决策。
- **架构的概念证明**。这个工作产品用于证明和确认架构的关键部分以降低项目的风险。它可能采用统一建模语言（UML）的解决方案模型或一个模拟的解决方案或一个可执行的原型。
- **变更要求**。这个工作产品描述了系统中的改变请求。这些改变可能很重大并影响迭代的关注点。
- **数据模型**。这个工作产品描述系统使用的数据的呈现方式。

- **部署模型**。这个工作产品列出了节点的配置、节点之间的通信和部署在节点上的组件。
- **功能模型**。这个工作产品描述了软件组件，包括它们的功能和相互之间的关系，以及在提供要求的功能时组件之间的协作。
- **RAID 日志**。正如我们在上一个列表中所述，这个项目管理工作产品记录了项目的风险（risk）、假设（assumption）、问题（issue）和依赖条件（dependency）。
- **复审记录**。这个工作产品定义架构复审的结果（在这项活动的上下文中）。
- **软件架构文档**。这个工作产品通过好几个架构视图来描述系统的不同方面，从而提供了一个全面的系统架构概述。

8.1 从需求走向解决方案

这一章是我们第一次考虑把需求转变为解决方案。图 8.2 显示了架构师在转变的过程中需要考虑的部分工作产品，并帮助我们演示一些简单却重要的概念。

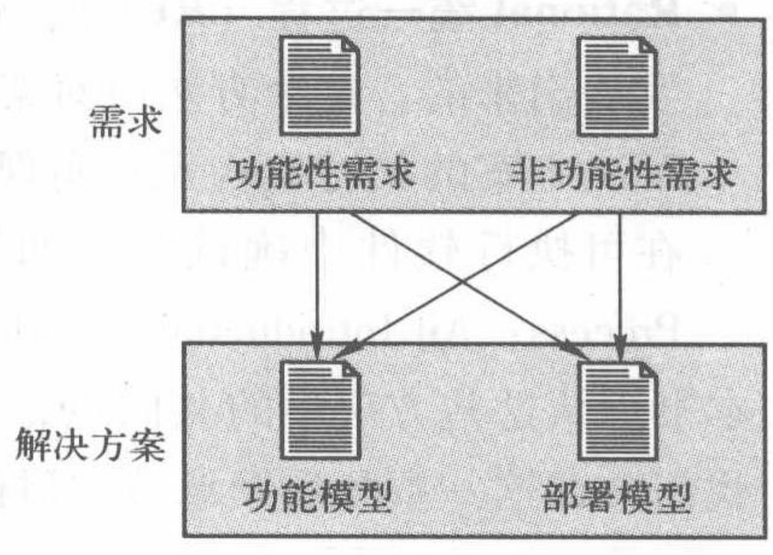


图 8.2 从需求移到解决方案中的关键工作产品

在这个图中，您看到了两个关键的需求工作产品：**功能性需求**和**非功能性需求**。您还看到了两个关键的解决方案工作产品：**功能模型**和**部署模型**。图中的箭头（它指出了各种影响）描述了一些重要且显而易见的概念：

- 功能性需求影响**功能模型**。例如，一个允许下订单的用例要求在解决方案中有合适的功能性元素来提供这个功能。
- 功能性需求影响**部署模型**。例如，一个在参与者（代表人）和系统之间的交互将要求解决方案中使用一个用户工作站。
- 非功能性需求影响**功能模型**。例如，一个必须提供某些功能的特定第三方包将成为一个强制的约束。
- 非功能性需求影响**部署模型**。例如，一个可靠性的需求会引入额外的节点来实现在一个系统元素失效时进行故障转移。

尽管这些单个的说明很容易理解，但是，同时考虑这些说明并确保**功能模型**、**部署模型**和其他工作产品保持一致却不容易。我们在本书中采用分而治之的方法，依次考虑这些内容的同时确保这些架构模型保持一致。

有不同的方法可以用来获得一个架构解决方案，但它们对图 8.2 中描绘的所有元素的考虑都是变化的。一些方法关注功能性需求，甚至达到排斥相应的非功能性需求的程度。其他的方法关注功能性解决方案的元素而不考虑同等重要的部署环境。本书同等强调功能性需求、非功能性需求、功能元素和部署因素，因为作者们相信任何的不平衡都会导致产生的架构不理想。

有好几种方法可以获得架构，也有好几种方法值得注意。《Generalizing a Model of Software

Architecture Design from Five Industrial Approaches》（Hofmeister 2005）中讨论了好几种方法的共同性，其中包含了5种方法。下面列出其中的3种方法：

- **属性驱动设计（ADD，Attribute-Driven Design）方法**，这个方法是由卡内基·梅隆软件工程研究所研究出来的。在这个方法中，通过使用满足质量属性场景的架构策略和模式，质量属性（例如可靠性）用于驱动架构和设计的演化并用于接下来的各级分解中。如果想要了解更多的信息，请查看《Software Architecture in Practice，2nd ed》（Bass 2003）。策略在后面的“概念：策略”部分进行了讨论。
- **西门子的4重视图（S 4V）方法**，这个方法是由西门子研究院研究出来的。这个方法从整体分析影响架构的因素开始，例如功能性需求、期望的系统质量、组织的约束和技术约束等。反复地应用4种视图（概念、运行、模块和代码架构）来确认架构的关键挑战和解决这些挑战的策略。如果想要了解更多的信息，请查看《Applied Software Architecture》（Hofmeister 2000）。
- **Rational统一过程（RUP）**，这个方法是由Rational软件公司（现在是IBM Rational）发展出来的。这个方法由对架构有影响的用例、非功能性需求和风险来驱动。在利用解决方案的关键架构元素实现需求之前，每次迭代都考虑这些关键元素。解决方案是在可执行软件中确认的。如果想要了解RUP的概况，请查看《The Rational Unified Process：An Introduction，2nd ed》（Kruchten 2000）。

本书承认这些方法中的共同点，也承认我们看到的成功的软件架构师遵循的那些最佳实践。我们会在这一章和后续章节中讨论这些实践。

概念：策略

策略的概念是属性驱动设计（ADD）方法的关键组成部分，《Software Architecture in Practice，2nd ed》（Bass 2003）中对它进行了详细的阐述。策略是决定如何满足一个功能性需求或如何达到一个质量属性的设计决策。例如，一个处理可靠性的策略可能是在系统中引入冗余。

本书中没有提供策略的综合分类，因为这样做会导致本书成为覆盖许多可重用资源（参考架构、架构模式、设计模式等，这些可重用资源都是围绕一个或多个策略构建的）的宏篇巨著。作为代替，我们关注方法框架，它考虑的大体上是可重用资源（包括策略）的应用。除了很多关于不同模式的书籍之外，《Software Architecture in Practice，2nd ed》（Bass 2003）和《Software Systems Architecture：Working with Stakeholders Using Viewpoints and Perspectives》（Rozanski 2005）都为那些架构师用来解决特定挑战（如可靠性、性能和可用性等）的策略提供良好的基础。

8.2 逻辑架构的价值

考虑逻辑架构的一种方法是把它看作一种帮助我们尽可能快地把需求变为可以用代码实现

的物理（和特定技术的）架构的简单有效的方法。在这种方法中，逻辑架构本质上是可抛弃的。另一种方法是您也可以把逻辑架构本身看作是战略性的资源从而进行积极地维护。在这种方法中，如果想用新的技术重新设计您的系统，逻辑架构可以提供一个系统的极好的抽象。在许多项目中采用的方法介于这两种方法之间，只有部分项目创建并维护逻辑架构。在这一部分将讨论这些考虑。

8.2.1 使逻辑架构最小化

如果逻辑架构仅仅是一种能够尽快到达物理架构的方法，您只需要产生足够的逻辑架构来证明您理解了当前的需求且这些需求已经适当地分配到解决方案中。

在极端情况下，根本不需要逻辑架构。在某些情况下这是有可能的——例如，当系统的需求和那些已经存在的系统类似的时候。在这些情况下，您可以通过查看现有的系统来得到架构的主要元素。另一个例子是包含一个封装的应用或现有的系统集成，需要的东西已经存在，因此，可以把逻辑架构的需要最小化。

另一个因素是系统的复杂性。例如，当您预见到系统非常分散且在逻辑层考虑部署将有助于您在物理层理解部署元素时，您可以选择开发一个**逻辑部署模型**。

8.2.2 把逻辑架构作为一项投资

除了把逻辑架构作为从需求到物理架构的有价值的垫脚石之外，在预见到技术变革的时候，逻辑架构还可以维护以提供一个良好的开端。您能够使用当初的逻辑架构（假设系统需求没有改变），不会因为技术变革而必须重新发明创造，因为总体来讲，逻辑架构是独立于技术的。

与不强调逻辑架构相对的是把逻辑架构看作是对技术变革有免疫力的投资。因此，从投资的角度来看，在预计到未来能够支持系统重新设计的技术变革时，维护逻辑架构是有意义的。

在决定是否把逻辑架构看作是需要不断维护的东西时，请考虑下列问题：

- 系统大且复杂吗？它的开发可能会持续很长时间吗？周期长的项目更有可能从逻辑架构的投入中得到回报。
- 系统需要好几个版本来发展并在它的生命周期中可能需要适应技术上的变革吗？在进行技术上的变革时，参考独立于技术的架构描述经常是有用的。
- 有开发和维护逻辑架构的能力吗？主要由开发人员或程序员组成的组织可能没有从架构上进行思考的人。如果由于没有能力而不能开发或维护逻辑架构，最初的投资成本（培训或雇用的）将很难证明是恰当的，构建这样的一个架构简直是不可能的。

8.2.3 可追溯性的重要性

无论您采用哪种方法，理解如何从一个工作产品得到另一个工作产品很重要，这种知识能

够让您对改变一个需求或架构元素所造成的影响进行分析。补充内容“概念：可追溯性”阐述了可追溯性的含义和其重要性。

作为项目的架构师，您需要考虑在开发过程中把可追溯性描述到什么程度，还要考虑可追溯性是显式的（例如添加特定的支持可追溯性的建模元素，例如需求实现，我们将在这一章后面讨论）或是隐式的（例如用实现的架构元素来命名代码元素）。

概念：可追溯性

可追溯性是用于把一个工作产品中的元素连接到相同或不同工作产品中的元素的一种机制。从开发生命周期来看，可追溯性使架构师能够从需求追溯到逻辑架构再到物理架构再到代码。可追溯性使您能够对改变一个需求所造成的影响进行分析，或分析优化一个程序算法对整个系统的影响。通过考虑如何（显式地或隐式地）在您创建的工作产品中描述可追溯性来同时支持这两种场景。

8.3 流程应用

支持架构规范的任务会随着整个开发生命周期发生变化，如图 8.3 所示。

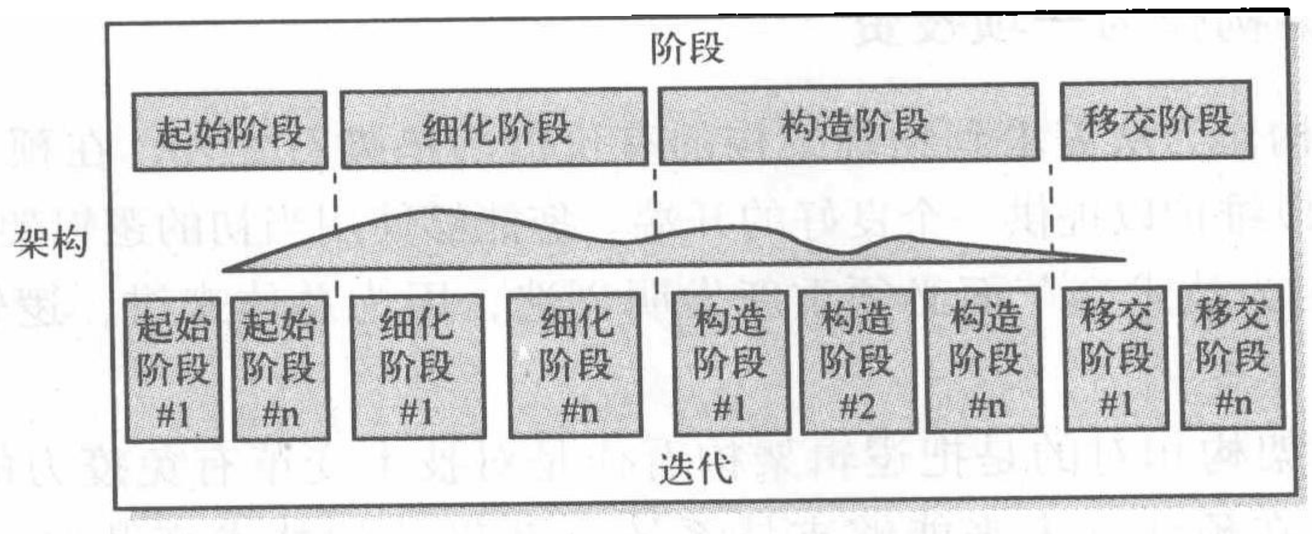


图 8.3 逻辑架构和迭代式开发

在细化阶段见到的逻辑架构任务最多，在那里您关注的是确定软件架构的基线。在这一阶段中，您的注意力将集中在分析那些认为对架构重要（也就是对架构最有影响）的需求上。在构造阶段，您分析剩下的需求，但是，这项工作不如在细化阶段中进行得那么广泛，因为大多数主要的系统元素已经发现，而且主要的技术决策已经确定。由于您必须分析的需求的数量减少和工作重心向开发方向转移，您花在逻辑架构上的时间将在构造阶段中逐步减少。在移交阶段，您可能还需要执行少量的逻辑架构任务，因为在把系统交付给用户时会根据接收到的反馈而引入需求变更，尽管出现这种情况的可能性不大。

8.4 创建逻辑架构：活动概览

图 8.4 显示了组成**创建逻辑架构**活动（在第 1 章中介绍的）的任务。

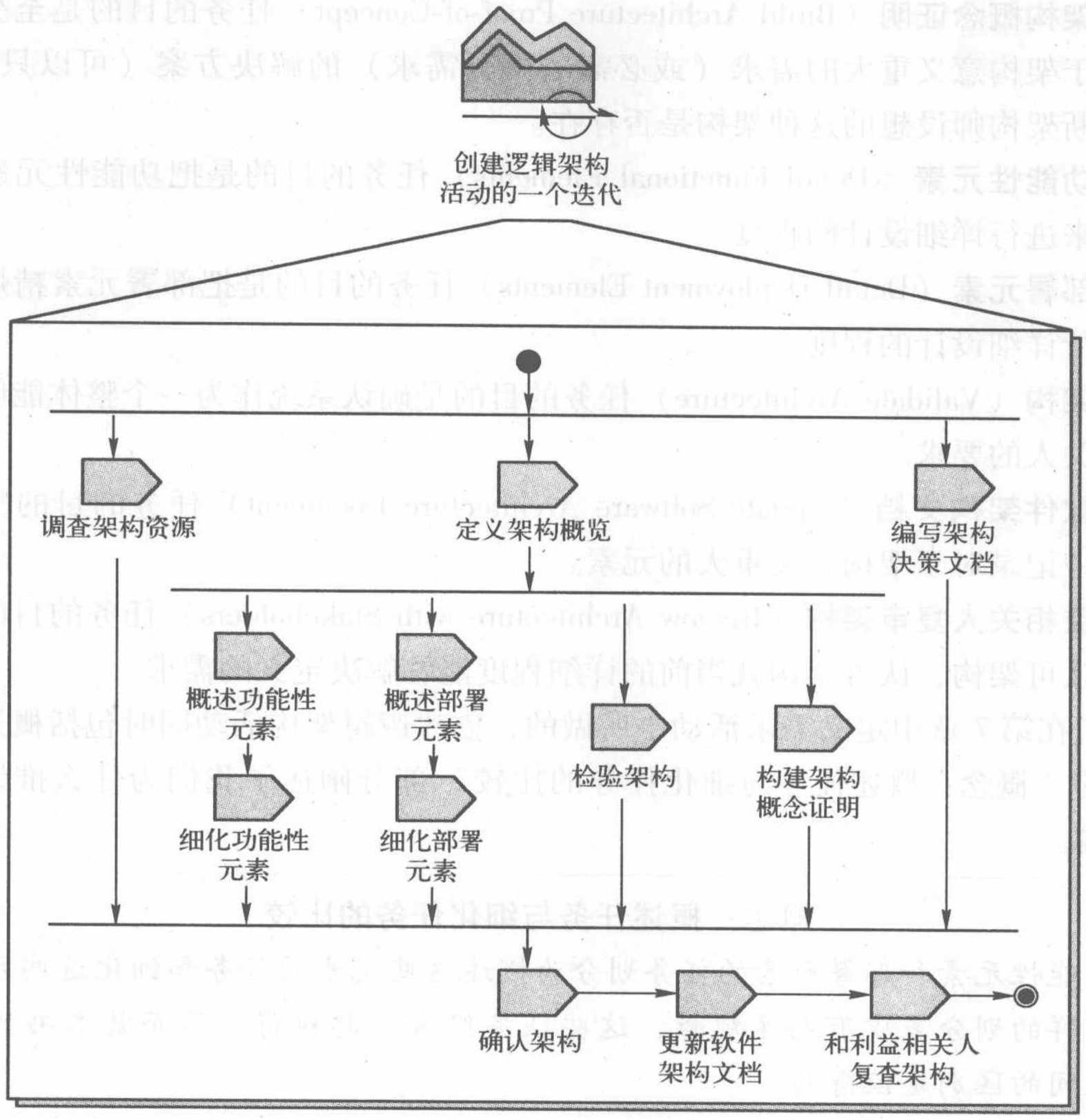


图 8.4 创建逻辑架构活动的概览

这些任务可以总结如下：

- **调查架构资源**（Survey Architecture Assets）任务的目的是确认可用于所开发系统的可重用架构资源。
- **定义架构概览**（Define Architecture Overview）任务的目的是确认并描述所开发系统的主要元素。
- **编写架构决策文档**（Document Architecture Decisions）任务的目的是记录架构形成过程中所做的关键决策和决策背后的原因。
- **概述功能性元素**（Outline Functional Elements）任务的目的是确认所开发系统的主要功能性元素（子系统和组件）。
- **概述部署元素**（Outline Deployment Elements）任务的目的是确认所开发系统将要部署的位置和各个位置内的节点。
- **检验架构**（Verify Architecture）任务的目的是检查架构工作产品是否一致并确保跨架构工作产品的关注点（如性能质量）已经一致地解决。

- **构建架构概念证明**（Build Architecture Proof-of-Concept）任务的目的是至少合成一个满足对于架构意义重大的需求（或必需的部分需求）的解决方案（可以只是概念性的）以判断架构师设想的这种架构是否存在。
- **细化功能性元素**（Detail Functional Elements）任务的目的是把功能性元素精炼到可以交付来进行详细设计的程度。
- **细化部署元素**（Detail Deployment Elements）任务的目的是把部署元素精炼到可以交付来进行详细设计的程度。
- **确认架构**（Validate Architecture）任务的目的是确认系统作为一个整体能够支持各种利益相关人的要求。
- **更新软件架构文档**（Update Software Architecture Document）任务的目的是在软件架构文档中记录对于架构意义重大的元素。
- **和利益相关人复审架构**（Review Architecture with Stakeholders）任务的目的是使利益相关人认可架构，认可架构就当前的详细程度能够解决定义的需求。

正如我们在第7章中定义需求活动中所做的，创建逻辑架构活动同时包括概述任务和细化任务。后面的“概念：概述任务与细化任务的比较”部分阐述了我们为什么推崇这个方法来获得一个架构。

概念：概述任务与细化任务的比较

关注功能性元素和部署元素的任务划分为概述这些元素的任务和细化这些元素的任务。实际上，这样的划分多少有些主观性，这些任务经常一起执行。然而就本书的内容而言，理解两者之间的区别是值得的。

实际上，概述任务关注对架构最有影响的元素和由项目首席架构师负责的元素。例如那些对与非功能性需求有关的解决方案进行重要决策的任务。

细化任务关注那些虽然重要但对理解整个系统的架构影响较小的元素。细化任务通常是关注系统某些方面的架构师的职责，如系统架构师、基础设施架构师和数据架构师。

表8.1总结了本章阐述的概述任务和细化任务中的元素。

表8.1 在概述任务和细化任务中考虑的元素

元　素	概　述	细　化
功能性	子系统	接口
	组件	操作签名
	操作	映射到数据
		操作的先决条件和后置条件
部署	地点	部署单元
	节点	连接

任务：调查架构资源

目　　的

这项任务的目的是确认可用于所开发系统的可重用资源。

角　　色

应用架构师（次要的）、数据架构师（次要的）、基础设施架构师（次要的）、首席架构师

输　　入

架构概览、部署模型、现存 IT 环境、企业架构规范、功能模型

输　　出

架构决策

步　　骤

调查架构资源。

架构师角色

架构师负责这项任务。

正如在第 5 章中所述，架构师灵感的一个关键来源是考虑可用的可重用资源。

> 设计专家知道不能根据基本原则来解决每一个问题。相反，他们会重用过去成功的方案。当他们发现一个好的方案时，他们会反复地使用它。这种经验是他们成为专家的部分原因。（Gamma 1995）

这项任务在一次迭代中通常会执行好多次，因为架构资源的重用始终是架构师头脑中首先考虑的问题，正如我们在第 5 章中详细讨论的。例如，您在执行这项任务时可能按顺序创建架构概览、功能模型和部署模型，因为这些工作产品已经确认能够由现有的资源满足的元素。

步骤：调查架构资源

当您开始软件架构的工作时，一个重要的考虑事项是复审这个领域中哪些工作已经做过，哪些能够在您的系统中重用。在第 5 章中讨论了各种类型的架构资源和这些可重用元素如何按照许多属性进行分类。这些属性可以提供一些关于一个资源在架构定义的不同阶段的适用性的线索。

资源的重用程度依赖于架构师工作的上下文。在早期的迭代中，架构师会选取对架构有深远影响的资源。一个典型的例子是选择一个定义系统主要元素的参考架构。架构师也可以选择把他们的架构基于一个封装的应用或重新设计一个遗留系统。同样地，这种选择对架构有显著的影响。资源选择的另一方面是选择一组特定的仅能用于**功能模型**或**部署模型**的模式。这些元素通常认为是和**概述功能模型**任务和**概述部署模型**任务一起考虑的。

其他可能相关的资源就是所谓的业界垂直模型，如 IBM 的保险应用架构（Insurance Application Architecture，IAA 2009），它提供了一套模型（包括数据模型和流程模型）来帮助保险公司有效、一致地应用信息技术。这些模型显然也会影响您创建的模型。

无论什么地方都需要强制使用的资源在企业架构层面就已经声明，在这种情况下，**企业架构规范**工作产品会包含一个或多个规范来指出这个事实。表 8.2 利用 Open Group 架构框架（TOGAF 2009）建议的属性展示了这样的一个规范样例。

表 8.2 企业架构规范样例

名　　称	购买而不是构建
声明	除非存在技术上或竞争上的原因，否则就购买业务应用和系统组件，而不是自己构建
原因	购买应用和组件通常可以赢得市场时间，而且能够获得比新构建的最初的系统更好的质量。购买的应用还可能提供更多的功能
隐含条件	需要把后续的支持费用考虑在内。 业务流程可能需要改变来支持现成的功能

由于许多开发工作都有棕色地带的性质，现有 IT 环境（记录在**现有 IT 环境**工作产品中）会提供一份包括已安装系统、数据、计算机和网络设施的清单——有可能其中的一部分或全部都包含在您的系统架构中（例如，也许系统已经在使用一个特定的数据库包）。因此，您要在这项任务中复审**现有 IT 环境**工作产品来判断是否存在某些元素可以由新系统重用。

调查架构资源的结果和决定使用哪些资源都是架构决策的特定形式，这些决策都会记录在**架构决策**工作产品中，我们会在这一章后面的**编写架构决策文档**任务中进行讨论。

任务：定义架构概览

目　　的

这项任务的目的是确认并描述所开发系统的主要元素。

角　　色

应用架构师（次要的）、数据架构师（次要的）、基础设施架构师（次要的）、首席架构师

输　　入

功能性需求、企业架构规范、术语表、非功能性需求、系统上下文

输　　出

架构概览

步　　骤

定义架构概览。

架构师角色

架构师负责这项任务。

架构概览提供了开发系统的一个高层描述。**架构概览**并不是一个可以用来构建系统的架构的完全说明；它更大程度上用于传递信息以使得系统利益相关者和开发人员在早期达成共识。

在项目的早期（例如当项目的初始原景描述出来的时候，或请求信息正发送给供应商的时

候），我们经常会开发出一些等同于**架构概览**的东西。在这种情况下，您至少应该参考这些图以确保项目与初始版本的连续性和可追溯性。

步骤：定义架构概览

架构概览触及好几个架构视图，但是在表达解决方案的主要构造块时主要趋向于关注功能视图和开发视图。如果在现阶段需要解决和澄清特定的关注点，那也需要提供其他的视图，如系统管理和安全。它还包含早期工作中对于解决方案的假设条件并经常用作为系统的非正式草图。接下来的任务（在这一章后面讨论）会导致创建更详细和更精确的工作产品，如**功能模型**和**部署模型**。

架构概览本质上通常是结构性的并倾向于显示为方块和直线，直线表示各种元素之间的依赖关系而不是交互。如果某些交互对于系统架构非常基本，则当然应该表示出这些交互。因此，**架构概览**实际上由许多图组成，不同的图表示不同的视图和不同利益相关者的关注点。

尽管我们没有推荐正式的符号来描述**架构概览**，但是，我们确实建议，无论使用什么符号，都要解释（连同一个关键词）并一贯地应用它们，例如一根末端是开放箭头的直线无论何时都表示相同的含义。这种方法不仅有助于理解，而且也能使利益相关者能够快速地领悟到概览想要描述的关键想法。

架构概览通常包含许多符号“样式”，而且常常包含方块（把一组功能表示成“组件”或“子系统”）和直线（把方块分开以表示“层”或“层级”）。子系统、组件、层、层级都是描述架构时使用的术语，有时候可以拥有不同的含义。补充内容“概念：子系统与组件”和“概念：层级与层”中讲述了这些术语在本书中的含义。

在必要的地方，**企业架构规范**工作产品中的规范应该在**架构概览**中得到体现。可能应用的典型的规范有：

- **重用**。这个规范表示在合适的地方必须重用已有的组件。
- **购买而不是构建**。这个规范在这一章前面的表 8.2 中已经进行了讨论。
- **单个视点**。这个规范表示把多个数据源组合起来，使所有相关的数据看起来像是来自单个数据源。
- **关注点分离**。这个规范表示组件应该负责一组关注相关业务的任务。

最后，虽然这项任务定义备选架构的方式不如接下来的任务那么正式，但是，它确认所有重要的需求，包括功能性需求和非功能性需求。特别是在非功能性需求方面，它代表了在处理系统要求的一些质量方面的初始考虑。它还包含了任何需求中声明的约束，例如任何现有环境带来的约束（例如需要在现有的网络环境中部署系统）、使用外部系统带来的约束（如需要和遗留系统集成）、使用指定的商业成品软件（COTS）带来的约束。

图 8.5 显示了 YourTour 系统的架构概览。通过复审**系统上下文**以理解 YourTour 系统的边界是什么、复审**功能性需求**以理解系统的功能领域是什么、复审**非功能性需求**以理解需要满足什么质量和需要符合什么约束，我们最终获得了这一概览。在这些工作产品中使用的术语应该

收集在**术语表**中。

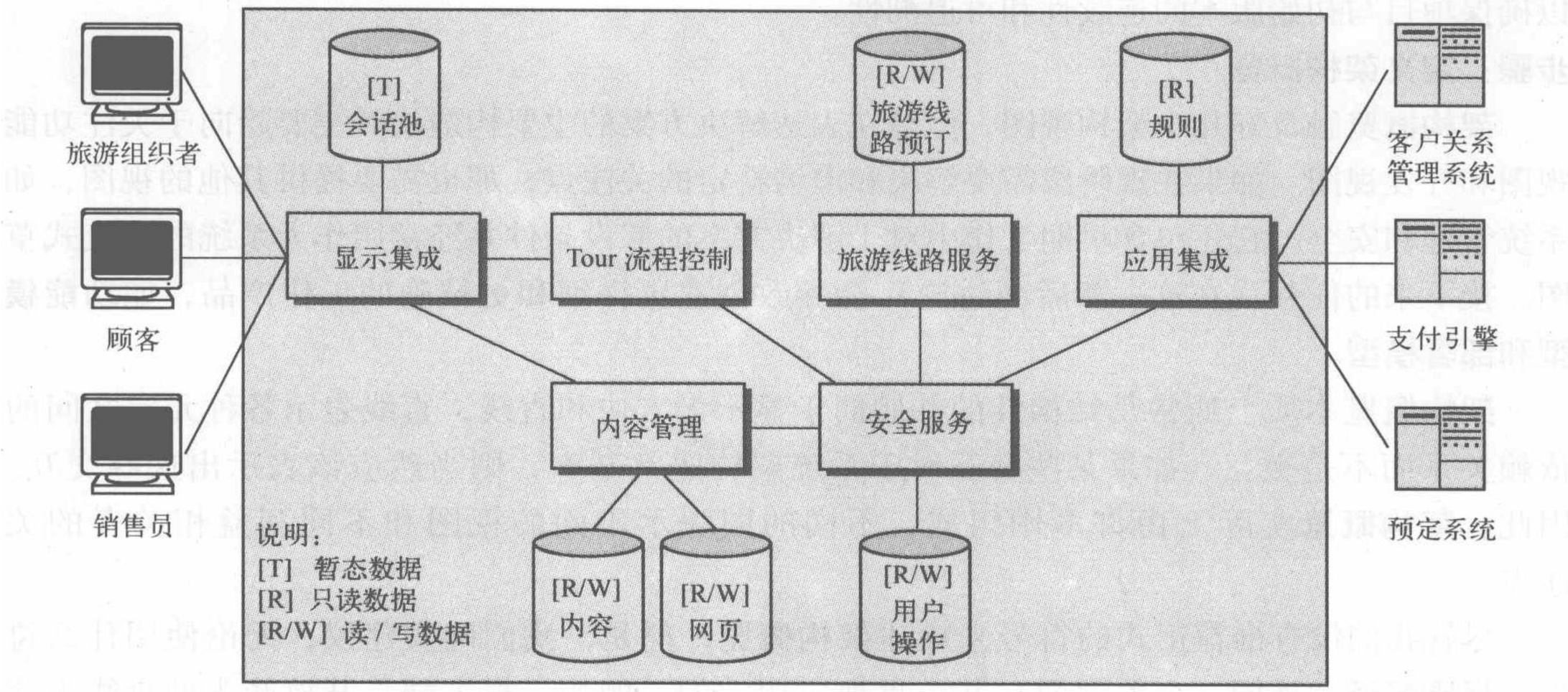


图 8.5 YourTour 系统的架构概览

概念：子系统与组件

子系统是一组相关的组件。在业务系统中，组件通常被组合在一起，因为它们共同支持某一主要业务领域（如客户关系管理或账户处理）。子系统也可以嵌套在一起，我们将在第 10 章中更详细地讨论这个话题，在那里我们将讨论系统的系统。

组件是封装自身的内容并在环境中可替换的一个系统模块化部分。一个组件根据提供和要求的接口来定义自己的行为。也就是说，一个组件作为一个类型，遵循这些提供和要求的接口（包括它们静态和动态的语义）。（UML 2.0 2003）

在创建图 8.5 中的**架构概览**时，我们使用了以下方法：

- 显示集成子系统把界面设备（计算机上的浏览器、移动设备等）和业务功能解耦，还把通常的业务服务转换为特定协议的用户界面。
- 我们相信有效的内容管理对于提高用户使用 YourTour 系统访问信息时的整体体验是重要的，如提供旅行保险。这些内容意味着需要包含一个专用的内容管理系统（Content Management System）来创建、编辑、管理、搜索和发布 YourTour 系统使用的各种数字化的内容（即图像、音频文件、视频文件、电子文档和 Web 内容）。
- 为了方便地对用户界面和 YourTour 系统的业务功能进行有效地解耦，我们提供了一个单独的流程控制子系统（Tour Process Control）。它负责控制任何业务处理逻辑的执行，还允许业务处理的变更独立于用户界面和业务服务（位于旅游线路服务子系统）。
- 安全性很重要，因为系统需要处理个人支付的详细信息，如信用卡信息和个人情况。

因此，需要一个专用的安全服务子系统来处理用户认证和授权。

- 我们希望能够把 YourTour 系统和错综复杂的遗留系统分开，因此，我们加入了一个专用的应用集成子系统来处理它们的交互。
- 最后，我们基于已经确认的子系统之间已知的（由我们已经采用的任何模式所表明的）或明显的交互来画出了它们之间端到端的连接。这些初始的端到端的连接会随着架构的演化而被精炼。

概念：层级（Tier）与层（Layer）

除了子系统和组件这两个概念，与系统结构分解相关的另外两个概念是层级和层。有时候这两个概念会出现在**架构概览**中。层级用于多层架构中，其中每个层级表示一组相关的处理逻辑。例如，典型的三层（tier）架构通常分为显示逻辑、业务逻辑和集成逻辑。一个层级代表的是对关注点的物理上的分离，在某种程度上也是一个逻辑分离，因为层级在物理上能够以不同的方式部署：

- 在单系统部署中，显示、业务和集成这三个层级都位于同一台物理机器上。
- 在瘦客户端部署中，显示层级位于客户端，但业务和集成层级位于服务器端。
- 在胖客户端部署中，显示和业务层级位于客户端，但集成层级位于服务器端。

层有时会和层级混淆，但它们是不一样的东西。相对于层级是物理元素的抽象而言，层的定义更为广泛，系统分层的概念遵循了层（Layer）架构模式，如同《Pattern-Oriented Software Architecture: A System of Patterns》（Buschmann 1996）和其他资料所述。简而言之，层模式根据组件的通用性（generality）来把组件放置于各个层中，通用性越高的组件（就是重用程度越高的组件）所放的层越低。简单来说，某一层中的组件可以使用同一层或它下层中的组件所提供的服务。如果组件仅仅能够和同一层或下面最近的一层中的组件通信，那就称作严格分层。如果组件能够和同一层或它下面任何一层中的组件通信，那就称为非严格分层。无论是哪种分层，组件永远都不应该和上层的组件通信，因为这种情况会降低组件的重用性。另外，从一个组件切换到另一个提供相同接口的组件也会变得困难。例如，图 8.6 中显示了一个从《Software Reuse: Architecture, Process and Organization for Business Success》（Jacobson 1997）书中得到的分层方法的样例。

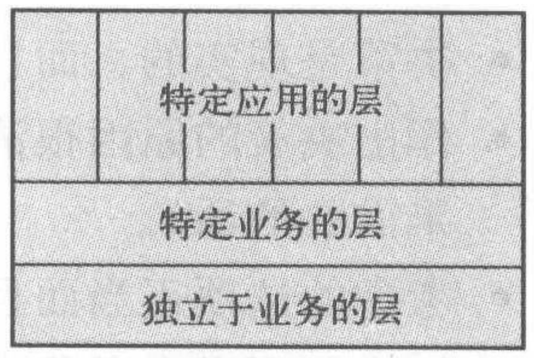


图 8.6 分层的样例

每一层的职责如下所示：

- **特定应用的层**包含特定应用的元素。这个图表明有许多应用。
- **特定业务的层**包含特定业务的元素或特定应用领域（如财务系统）的元素，但它们可以跨应用重用。
- **独立于业务的层**包含独立于业务或独立于应用领域的元素（如一个用户界面框架、消息总线、操作系统或数据库）。

任务：编写架构决策文档

目　　的

这项任务的目的是记录在架构形成过程中所作出的关键决策和背后的原因。

角　　色

应用架构师（次要的）、数据架构师（次要的）、基础设施架构师（次要的）、首席架构师

输　　入

所有和架构相关的工作产品、RAID 日志

步　　骤

- 记录难题和问题。
- 评估选项。
- 选择首选的选项。
- 编写决策文档。

架构师角色

架构师负责这项任务。

为了真正地理解一个 IT 系统的架构，理解形成架构的决策和这些决策背后的原因很重要。一些决策显式地记录在各种工作产品中（例如一个组件部署在一个特定节点这个事实）。然而，通常也会形成好几个隐式的决策，这些决策对整个系统的形成具有同样重要的影响，但是只有作出决策的人才知道。因此记录架构决策的目的是使架构师头脑中的东西变得更为清楚，并确保：

- 决策编写成文档以便利益相关者能够理解架构为什么会变成这样。
- 决策是现实的，而且基于实际的业务需要，而不是由技术驱动的。
- 作出决策的原因很清晰，能够让其他人复审和评估（包括内部和外部的审查人员，如果需要的话）。
- 考虑的选项正当而且合理。
- 架构师不必走老路来证明一个已经长期遗忘的决策。

正式记录的架构决策通常仅限于形成架构的关键决策，因此需要考虑其正式的程度。然而，即使在最简单的架构中，通常也存在好几个关键的决策影响了架构，而且这些决策对于必须理解或维护系统的人并不是显而易见的。因此，显式地把这些架构决策编写成文档是一个有价值的活动。

一些决策具有时效性，而且比项目周期更短。一个决策可能是短期的和战术性的以便用来推动项目，但是，在系统完成之前需要重新回顾它们。在这种情况下，需要对必须在指定时间进行复审的决策进行标注。相对地，一些决策是根本性的，它们实际的生命周期比产生它们的项目生命周期还长，因为它们适用于企业中的其他开发项目。这些决策可以看作是架构规范的

一部分并在企业架构层级进行应用。

如果想要了解更多关于架构决策的详细讨论，以及希望获得一个编写架构决策文档的模板，请参考论文“Architecture Decisions: Demystifying Architecture”（Tyree 2005）。

步骤：记录难点和问题

作出一个架构决策通常是因为确认了需要解决的难点或问题。难点通常记录在 **RAID 日志**工作产品中，或者由利益相关者或项目成员通过口头或电子邮件交流。类似地，作为风险提出的某些问题当然也会记录在 **RAID 日志**中。

作为编写架构决策文档的第一步，清晰、无歧义地记录需要考虑的难点和问题是有帮助的。这个文档可以是陈述难点的简单一句话，也可能需要较长的陈述来描述漫长的历史或上下文。在这个步骤中花些心思考虑一下以便弄清楚需要解决的问题，这是值得的。

步骤：评估选项

在这个步骤中，您要判断拥有哪些选项可以用来解决难点或风险。这个步骤可能需要参考模式（看看相似的问题是如何解决的）、搜索供应商的产品或库（是否可以获得一个产品或一个库来解决这个问题）或进行头脑风暴以得到其他的方法（如果问题涉及新的或挑战性的领域）。

步骤：选择首选的选项

在这个步骤中，您需要评估每个选项并决定采用哪一个。在决定采用哪一个选项的时候，您必须考虑下面几个问题：

- 有没有任何功能性需求、质量要求或约束能表明一个选项比另一个选项更好？例如，成本或项目进度是重要的因素吗？
- 您必须遵循任何规范、政策或指导吗？例如必须遵循企业范围的政策。
- 采用某些选项会比采用其他选项更有风险吗？例如，它们使用了新的或没有得到证明的技术吗？如果是，项目或组织是否愿意承担这些风险？
- 这些选项是策略性的还是战略性的？一些选项会产生策略性的短期成果而其他的选项则是长期的且具有战略意义吗？您也许能够通过在现有机器上添加处理器等硬件资源（即采用垂直的扩展方式）的方式来使系统的性能获得短期的改进。但从长期考虑，添加机器是更好的办法（即采用水平的扩展方式）。
- 实际上存在多种选项由您选择吗？有时候您的情况可能并不是简单地选择一个选项。短期（策略性的）的和长期的（战略性的）选项都是适用的。
- 一些决策具有依赖性，或者依赖其他的决策，或者依赖用以验证决策的原型的运行结果。如果这些依赖性确实存在，那您就应该注意这种情况。

步骤：编写决策文档

当作出决策后，对最初问题的陈述、考虑的选项、决策、导致这一决策背后的原因等都应该记录在**架构决策**工作产品中。表 8.3 中的例子来自 YourTour 系统的**架构决策**工作产品。

表 8.3 一个架构决策文档的样例

难点	YourTour 系统将应用少量的（可能少于 50）必须定期更新的业务规则。它的目的是允许业务用户（而不是开发人员）更新这些规则。这些规则将用于检查某些条件（例如根据以往的预定是否可以得到折扣）或执行计算（例如计算一个旅游线路的全部费用，包括所有的税和附加费）
架构决策	开发一个定制规则引擎组件，而不是使用一个封装的解决方案或将规则写入代码
假设	需要管理的业务规则数量相对较少（少于 50 条），并且变化频率较低（每年少于两次）
可替代方法	选项 1：在代码中嵌入规则 选项 2：开发一个定制规则引擎组件 选项 3：购买规则引擎库
选取的选项	选项 2
理由	在代码中嵌入规则是一个糟糕的方法，因为这不符合软件工程分解问题的原则，而且使规则难于维护。购买一个规则引擎的性价比不高，因为规则的数量很少且发生变化的频率也不高

任务：概述功能性元素

目　　的

这项任务的目的是确认所开发系统的主要功能性元素（子系统和组件）。

角　　色

应用架构师、数据架构师（次要的）、基础设施架构师（次要的）

输　　入

架构决策、架构概览、架构的概念可行性（可选）、业务实体模型、业务规则、企业架构规范、现存 IT 环境（可选）、功能需求、术语表、非功能需求、需求优先级列表

输　　出

功能模型

步　　骤

- 确认子系统。
- 确认组件。

架构师角色

架构师负责这项任务。

这项任务关注确认架构的功能性元素（子系统和组件），并确保得到的子系统和组件集遵循固定的架构规范。在本书中，我们使用**组件**这个术语来表示系统的主要构建模块。我们应当指出，正如在本书中定义的，组件不仅仅是技术上的软件元素（如 Enterprise JavaBean），而且也广泛地用于代表组成系统的架构元素。请参考我们在这一章前面的补充内容“概念：子系统与组件”，我们使用的就是这一定义。

当这项任务结束时，我们会定义出好几个结构良好的逻辑子系统和组件，以及它们的关系和职责。这些子系统和组件可能是用建模工具定义（例如支持创建 UML 模型的工具）的，或可能只是文档中的一个简单草图。在命名功能元素时，我们会使用**术语表**中定义的相关术语。

步骤：确认子系统

我们可以从**架构概览**工作产品中获得初始的一组子系统。图 8.7 是显示 YourTour 系统逻辑子系统和它们之间依赖关系的一个 UML 组件图样例，这也是 YourTour 架构的最高层视图。其中的每一个子系统都包含了一个或多个通过协作来支持系统要求功能的组件。

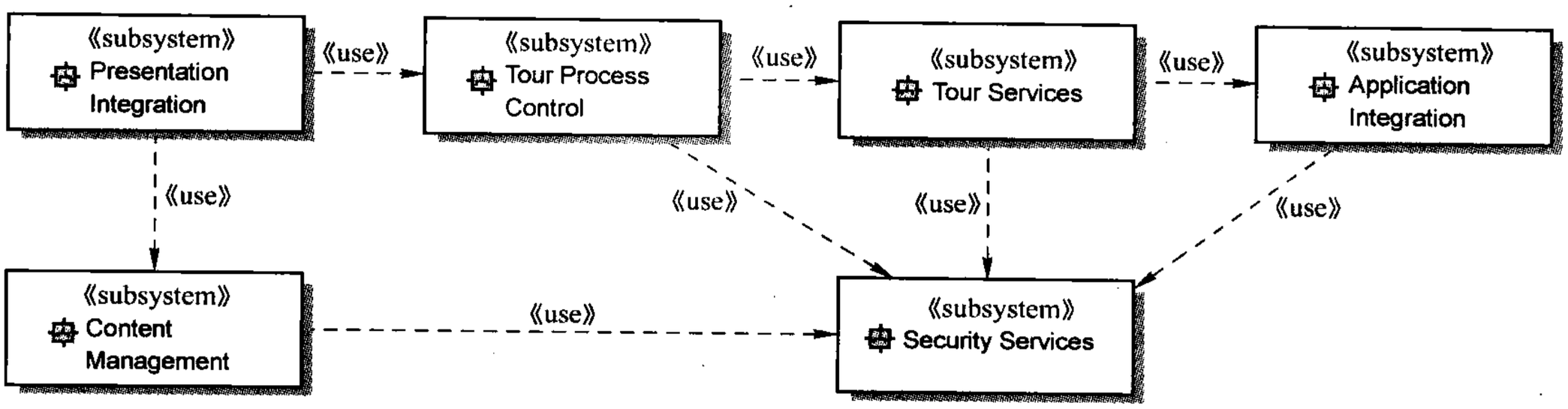


图 8.7 YourTour 逻辑子系统

我们简单地把**架构概览**中的非正式符号转换为 UML 就得到了图 8.7 中的内容。作为确认子系统的一项技术，这个技术和其他技术一样好用。如果当初我们没有为 YourTour 开发出如此详细的架构概览，那么，我们就必须使用以下技术来确认子系统：

- 复审**功能性需求**工作产品，寻找已经组合在一起或可以组合在一起的需求。例如，处理顾客管理方面的所有问题这个需求可能暗示了一个客户关系管理子系统。
- 复审**企业架构规范**、**非功能性需求**和**现有 IT 环境**工作产品，看看是否存在任何约束暗示着某些特定的技术（如应用服务器、消息代理、系统管理支持、事务处理程序等）。在架构层面，这些技术可以确认为分离的子系统。
- 复审**业务实体模型**，查找由一组相关组件进行管理的相关业务实体组。例如，把处理旅游线路的实体分为一组，这就意味着一个旅游线路服务子系统。

步骤：确认组件

当描述逻辑架构时，我们确认组件、组件的职责和组件之间的协作——这些都是由声明的需求所驱动的（后面的“概念：描述逻辑架构组件的特征”部分描述了逻辑架构组件）。Rebecca Wirfs-Brock 设计的 CRC（Classes-Responsibilities-Collaborations）技术使得这个方式变得流行。以这个技术为基础，我们考虑每个组件的以下方面：

- 组件知道什么（它拥有或管理的数据）。
- 组件做什么（它展示的行为）。
- 组件要求其他组件为它做什么（也就是它需要一起协作来执行某种功能的组件）。

组件的确认是建立在对**业务实体模型**、**功能需求**、**非功能需求**、**业务规则**、**现有 IT 环境**和**架构决策**等工作产品的分析之上的。在后面的讨论中我们会考虑每一个工作产品，在这些讨论中我们会阐述它们如何用于 YourTour 系统的**功能模型**的开发。虽然我们按照一定的顺序讨

论这些工作产品，但是，实际上我们趋向于把它们和相关的技术一起考虑。另外，我们会考虑已经创建的**架构概念证明**。

概念：描述逻辑架构组件的特征

一个逻辑架构包括系统的不同方面，表现为不同类型的组件。在很多情况下，在逻辑架构的开发过程中考虑三种类型的组件（Jacobson 1992）会帮助您按“分而治之”的方式完成您的工作：

- **边界（或显示）组件**。这些组件提供了系统和系统与之交互的外部事物（如最终用户或外部系统）的边界。边界组件通常调用控制组件。
- **控制（或执行）组件**。这些组件支持系统的控制逻辑，也支持业务规则和其他用以满足功能需求的逻辑。控制组件通常调用边界组件、实体组件和其他控制组件。
- **实体（或数据）组件**。这些组件支持持久数据的呈现。实体组件通常调用其他实体组件。

当采用 UML 来描述这些不同类型的组件时，您也可以选择使用一个包括每种类型组件模型的 UML 概图（profile）。使用的符号如表 8.4 所示。

表 8.4 逻辑架构中使用的组件类型

组件类型	UML 表示	描述
边界	<<Boundary>>	边界组件代表系统和系统环境之间的边界
控制	<<Control>>	控制组件代表系统的逻辑控制和协调
实体	<<Entity>>	实体组件封装了系统中呈现的信息

业务实体模型

通过把相关的实体分组后为每组实体分配一个管理的组件，**业务实体模型**可以用于确认组件。图 8.8 显示了 YourTour 研究案例的部分业务实体模型——它们负责管理和旅游线路相关的信息。在图中，旅游线路（Tour）、旅游线路类别（Tour Category）和旅游线路地点（Tour Location）等实体组合在一起，并分配给管理这些组件的旅游线路管理器（Tour Manager）组件。因此，您可以说旅游线路管理器组件的职责之一是管理和旅游线路相关的信息。当架构师对需求进行进一步分析并确定旅游线路管理器组件必须支持其他的操作来帮助系统满足功能性需求时，会有其他的职责分配给旅游线路管理器组件。旅游线路管理器组件位于旅游线路服务（Tour Services）子系统中。

功能性需求

分析有重大架构意义的**功能性需求**是确认组件的最稳健的方法之一。假设用例是您用来捕获功能性需求的主要方法，那您可以通过分析它们来确认组件，首先关注优先级最高的用例。正如在第 7 章“定义需求”中所述，**排定优先级的需求列表**决定了需求所具有的优先级。在讨论中我们使用了预定旅游线路（Book Tour）用例作为一个样例。预定旅游线路用例的上下文如图 8.9 所示（这个图就是第 7 章中的图 7.7）。

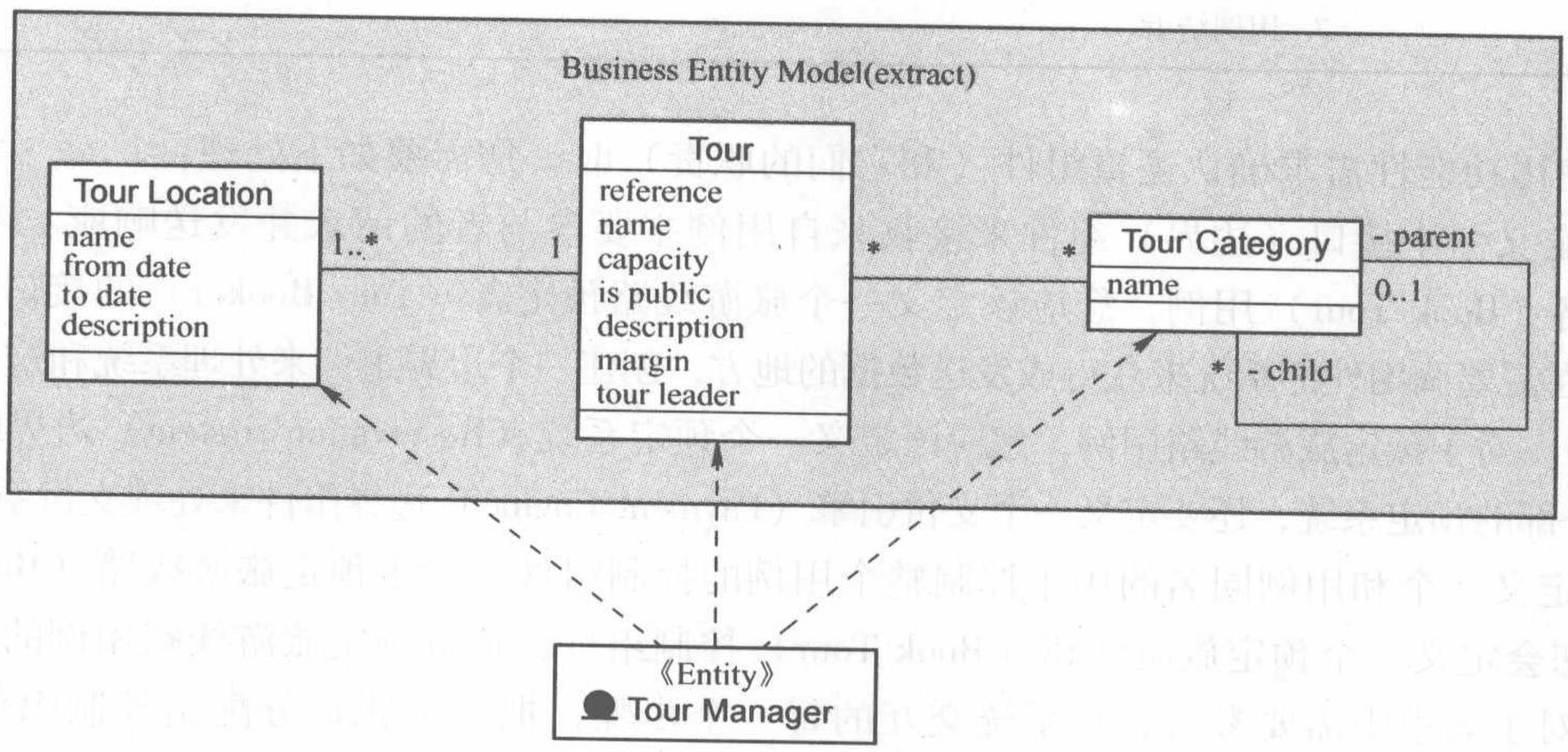


图 8.8　从一个业务实体模型中确认组件

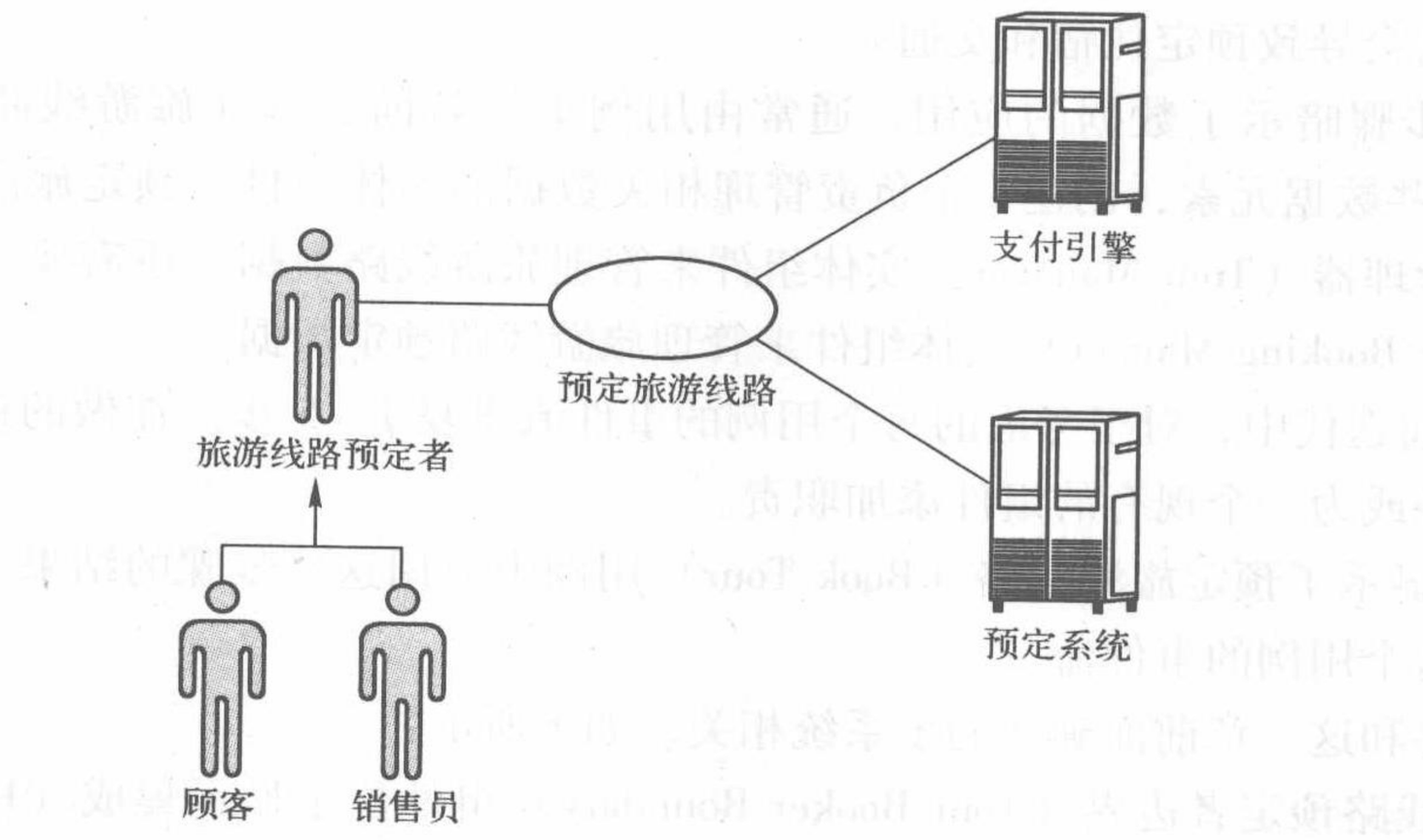


图 8.9　预定旅游线路用例的上下文

表 8.5（即第 7 章中的表 7.10）列出了预定旅游线路用例的主要事件流。当用这种方法分析用例时，取决于当前迭代的目标，您可能只考虑部分的事件流。这个例子仅考虑了预定旅游线路（Book Tour）用例中的主要事件流。

表 8.5 预定旅游线路用例的主要事件流

名　称	预定旅游线路
主要流	1. 这个用例起始于旅游线路预定者选择为一个或多个游客预定旅游线路 2. 旅游线路预定者确认要预定的旅游线路 3. 旅游线路预定者确认参加这个旅游线路的每位游客 4. 旅游线路预定者为每位游客指定他们想要的住宿和车程选项 5. 旅游线路预定者提供详细的支付信息。预定系统预定住宿和车程，支付引擎处理支付问题 6. 旅游线路预定者收到一个唯一的编号作为旅游线路预定的确认 7. 用例结束

当使用功能性需求确认逻辑组件（和它们的职责）时，您需要如下处理：

1. 定义一个接口（边界）组件来接收来自用例主要参与者的请求并发送响应。对于预定旅游线路（Book Tour）用例，您应该定义一个旅游线路预定者（Tour Booker）的接口组件。

2. 在需要调用外部系统来获得或发送数据的地方，创建一个边界组件来处理系统和这个外部系统的接口。对于预定旅游线路用例，您应该定义一个预定系统（Reservation System）边界组件来负责处理外部的预定系统，还要定义一个支付引擎（Payment Engine）边界组件来处理支付系统。

3. 定义一个和用例同名的用于控制整个用例的控制组件。对于预定旅游线路（Book Tour）用例，您会定义一个预定旅游线路（Book Tour）控制组件，负责预定旅游线路用例的执行。

4. 对于需求中需要参与者和系统交互的每一个步骤，把一个职责分配给控制组件来管理那个交互。对于预定旅游线路用例，您需要分配这些职责：获取一个有效旅游线路的列表并选取适当的旅游线路，把旅游参与者加入旅游线路，为每个旅游参与者添加旅行选项及获取支付详细信息（这会导致预定住宿和交通）。

5. 一些步骤暗示了数据的应用，通常由用例中的名词表示（旅游线路、旅游线路预定等）。对于这些数据元素，创建一个负责管理相关数据的实体组件。预定旅游线路用例需要一个旅游线路管理器（Tour Manager）实体组件来管理旅游线路数据，还需要一个旅游线路预定管理器（Tour Booking Manger）实体组件来管理旅游线路预定数据。

6. 在当前迭代中，对于考虑的每个用例的事件流重复 1 ~ 5 步。在做的过程中，您可能会确认新的组件或为一个现有的组件添加职责。

图 8.10 显示了预定旅游线路（Book Tour）用例中应用这些步骤的结果。元素之间的依赖关系也源自这个用例的事件流。

这些组件和这一章前面确认的子系统相关，如下所示：

- 旅游线路预定者边界（Tour Booker Boundary）组件位于显示集成（Presentation Integration）子系统之中。
- 支付引擎边界（Payment Engine Boundary）组件和预定系统边界（Reservation System Boundary）组件位于应用集成（Application Integration）子系统之中。
- 预定旅游线路控制器（Book Tour Controller）组件位于旅游线路处理控制（Tour Process Control）子系统之中。

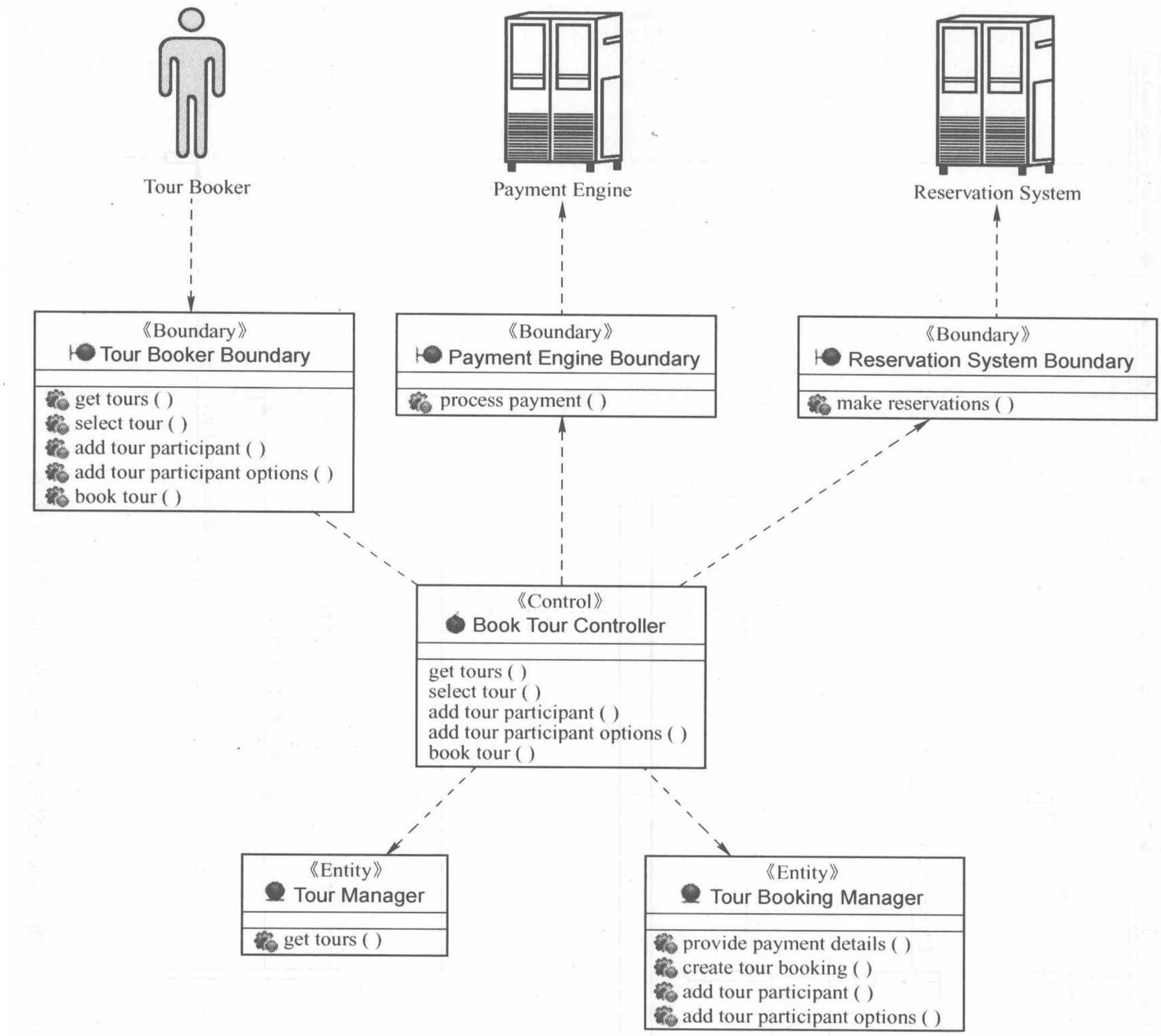


图 8.10　来自预定旅游线路用例的组件

- 旅游线路管理器（Tour Manager）组件和旅游线路预定管理器（Tour Booking Manager）组件位于旅游线路服务（Tour Services）子系统之中。

组件通常分配给子系统，因为它们在功能上是相关的并因此创建一个耦合的子系统（这就是把旅游线路管理器组件和旅游线路预定管理器组件组合在一起的原因），或因为它们在技术上是相关的，并因此创建一个可能部署在同一个运行平台上的子系统（这就是把控制组件、系统接口和用户界面组件分配在一起的原因）。

您可以通过对组件之间的交互显式地建模来发现每个用例中的事件流，从而得到组件另外的职责。结果就是把一些职责分配给响应一个特定消息的一组组件。图 8.11 是预定旅游线路（Book Tour）用例的主流程的时序图，它显示了组件如何通过协作来满足这一章前面的表 8.5

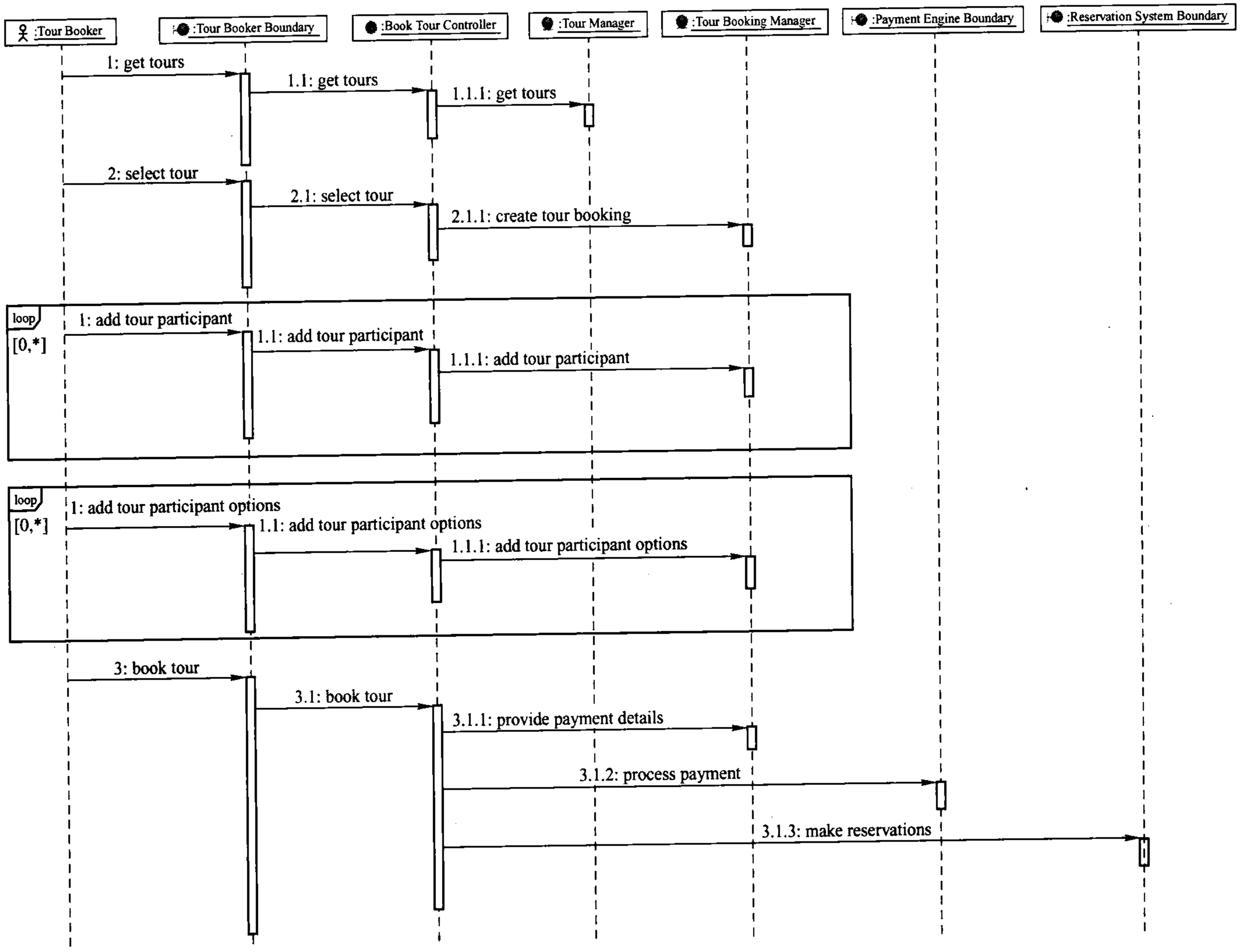


图 8.11　预定旅游线路用例（主流程）的 UML 时序图

中描述的用例。

确保组件能够追溯到产生它们的需求，这也是一个很好的做法。补充内容“最佳实践：追溯方案元素直至需求”中描述了如何利用 UML 做到这一点。

我们要强调的是，这种方式会导致我们在确认组件和它们的职责时进行第一步划分。组件可能需要重构好几次以确保它们的架构设计得很好。虽然没有确定的度量标准来衡量制定结构良好组件的方法的优劣，但是，遵循高内聚和低耦合的原则以及确保按适当的粒度级别定义组件（如在后面的“概念：组件质量测量”部分讨论的）都是确保组件构架良好的方法。

最佳实践：追溯方案元素直至需求

在考虑需求和它们在解决方案中如何得到满足的时候，显式地从解决方案中的元素追溯到它们要解决的需求是有益的。这种可追溯性可以帮助您确定改变需求的影响。图 8.12 中显示了表明这种可追溯性的一个需求实现的概念。需求实现是一种构造型的 UML 协作，它仅仅指明哪些方案元素协作来满足一个特定的需求。在图 8.12 中，实现预定旅游线路用例的方案元素是包含在参与者（Participants）UML 组件图中的元素和显示在主要事件流（Main Flow of Events）UML 时序图中交互的那些元素。图 8.12 中以 UML 标注的形式显示了这两个图的引用。按照惯例，这个需求实现的名称和它实现的需求的名称相同。

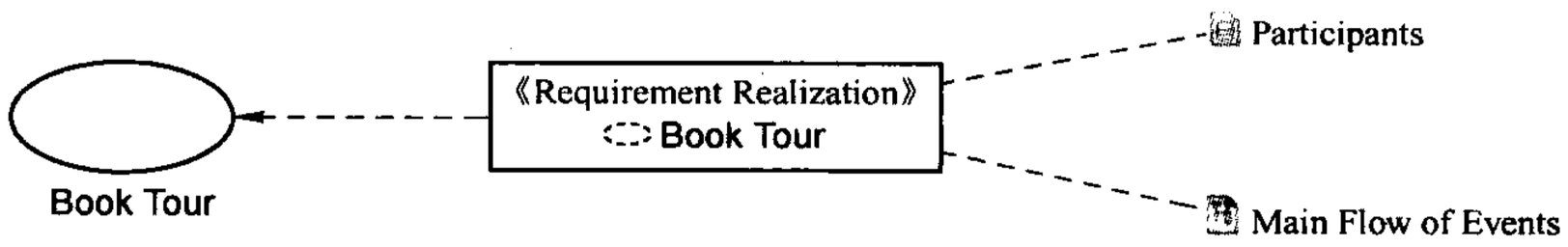


图 8.12　预定旅游线路用例的需求实现

需求实现的概念也可用于非功能性需求的实现，正如我们在接下来的部分中所述。另外，一个需求实现可以参考显示部署元素（而不是功能元素）和部署元素如何实现功能性需求和非功能性需求的图。

建立和维护从架构到需求的可追溯性可能是费时的。对于相对较小的项目，您可以利用电子表格来开发一个可追溯性表格。然而，使用这种方式定义的可追溯性可能难于维护。一个更好的、适用于大型或复杂项目的方法是使用一个适当的开发工具，这个工具能够明确地支持在方案元素和它们实现的需求之间进行追溯并维护这些信息。

概念：组件质量的度量

软件的内聚性和耦合性质量（关于应用于计算机程序的）作为降低软件维护和修改的成本的一个方法，是由 W. P. Stevens（Stevens 1974）首先提出来的。

内聚性是组件的职责之间的相关程度的一个度量。高内聚（一个组件拥有一组紧密相关的职责）认为是好的；低内聚（一个组件拥有一组相关性差的或不相关的职责）则认为

是差的。高内聚之所以好是因为它使得架构易于理解。系统维护也可能更容易，因为更改仅限于一个单独的组件，而不会遍布整个系统。如果组件具有高内聚性，则更可重用，因为一组紧密相关的功能作为一个整体来使用的可能性比一个组件提供的一组随意的功能更大。

耦合与内聚有关，它表示组件之间关联的强度。强耦合的组件具有紧密绑定的关系，组件之间高度依赖。弱（或松）耦合优于强（或紧）耦合，因为从一个组件切换到另一个组件或更新一个组件而不影响其他组件会更容易。低耦合的组件比高耦合的组件更有重用的可能性，因为根据定义，它们可以更轻易地从一个系统中抽取出来，然后插入到另一个系统中去。

粒度是质量的另一个度量。它是对分配给组件功能的大小和数量的衡量。架构师在评估组件的粒度之前必须理解组件所处的环境。一个极端情况是，整个系统都描绘成单个组件，而且在系统支持的业务流程环境中可能完全合适。另一个极端情况是，系统由许多细粒度的组件组成，在特定环境中这也是有可能合适的。《Business Component Factory》（Herzum 2000）中对一个组件可能存在的不同粒度进行了有趣的讨论。本书不仅讨论了简单的工具组件，也讨论了设计来支持分布环境的组件和元素跨越分布环境中不同层的业务组件。

非功能性需求

如此严格的考虑功能性需求仅仅是工作的一半，因此，有那么多的开发方法竟然到此为止令人惊奇。这经常导致一个系统功能正确却无法提供必需的质量。另一个有助于架构师理解所需组件和它们特征的来源是系统的非功能性需求，它们记录在**非功能性需求**工作产品中。在这一部分，我们既考虑系统必须具备的质量（如性能和可靠性），又考虑系统必须适应的约束（如强制性的技术、第三方组件的应用和遗留系统的使用）。

这些考虑可能会导致确定新的组件。YourTour 系统有一个可测试性的需求——“系统允许记录所有的事务内容以便所有的事务和它们的有效负载都可以验证”，这就确定了一个事务日志组件并加入到旅游线路服务（Tour Service）子系统。同样地，YourTour 系统还有一个集成约束——“YourTour 系统将使用 YourTour 现有的客户关系管理系统（CRM）来存储客户信息”。这个约束导致增加一个组件来管理 YourTour 和 CRM 系统（将纳入应用集成子系统）的接口。图 8.13 显示了这些例子确定的组件。

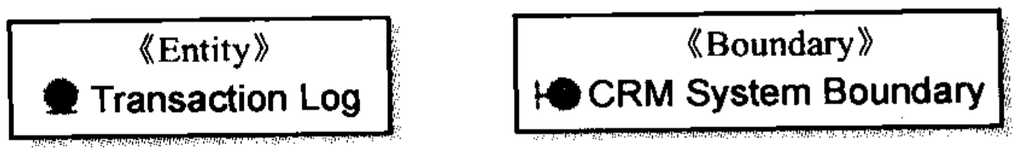


图 8.13 从 YourTour 非功能性需求获得的组件

考虑非功能性需求可能不仅导致确认新的组件，还会指明某个特定的组件（或者，更可能的是一组组件）需要满足特定的质量，从而导致您精炼组件，同时还可能相应地修改组件说明。特别地，您可能得到应用于一个特定组件的非功能性需求。为了做到这一点，您需要依次查看每一个非功能性需求并基于已经确认的组件把需求分配给它们。表 8.6 中列出了一个例子。

表 8.6 把非功能性需求分配给组件

需求	组件	操作	需求的允许时间
从发出请求到用户得到确认，预定一个旅游线路的操作必须小于 10 s	旅游线路预定管理器组件	provide payment details( )	1 s
	支付引擎边界组件	process payment( )	3 s
	预定系统边界组件	make reservations( )	6 s

如果您正在使用 UML 记录这些额外的信息，创建一个允许记录这些附加信息的 UML 概图是合适的。UML 概图是一种扩展机制，它允许在标准的 UML 建模元素中加入额外的属性（基于应用 UML 构造型的基础上）。

当把非功能性需求分配给组件的时候，请注意每一个组件都要能够支持规定的需求，尤其是当一个组件可能依赖一个外部系统来实现它的职责的时候。在表 8.6 中，小于三 s 处理支付的需求依赖于支付引擎（一个您可能无法控制的外部系统）能够在给定的时间内完成处理的能力。架构师需要理解有哪些组织内部的协议或者甚至是商业的协议来管理和监控这种情况。

设计一个解决方案来应付可能存在冲突的非功能性需求，这包括转变相反的力量，正如我们在第 2 章中讨论的。通常会存在两个或多个解决方案，每个方案中都包括在成本、可行性等方面的利弊。当考虑采用哪种方案时，用**架构决策**工作产品来记录所有的选项和选择背后的原因是有帮助的。

业务规则

业务规则的考虑，正如记录在**业务规则**工作产品中的，可能还会导致组件的确认。在 YourTour 系统中，好几项业务规则都和旅游线路的管理及如何预定它们有关。假定这些规则会随着时间而改变，这是合理的，因此您可以选择把这些规则分离到一个规则组件中，我们把这个组件放在旅游线路服务（Tour Service）子系统中。然后，位于那个子系统中的所有组件都利用这个规则组件来访问当前的规则，如图 8.14 所示。

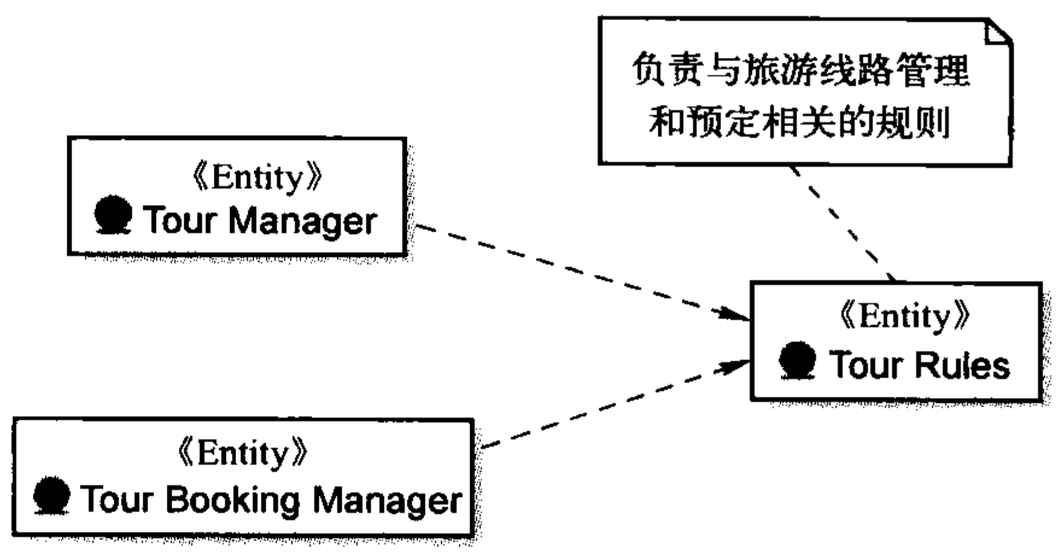


图 8.14 YourTour 业务规则组件

对于业务规则的放置和描述的讨论，请看后面的“概念：业务规则的放置和描述”部分。

概念：业务规则的放置和描述

以下问题有助于确定业务规则的放置：

- **业务规则有可能频繁地改变吗？**答案指出了需要多大的灵活性更新业务规则。对于一个有可能频繁更改的规则，最好将它分配给一个专用于实现这条业务规则的组件。
- **业务规则应用于很多场景，还是仅针对一个特定的用途？**答案指出了这个规则将赋予哪个组件或哪些组件。对于需要在很多场景中重用的业务规则，最好把这个规则

赋予一个单独的组件，并相应地调用它。对于一个仅仅用于一个场景的业务规则，最好把这个规则置于要求执行这个业务规则的组件中。

- **系统用户需要业务规则具有可见性吗？**如果是，您可能需要允许业务规则的某些方面在用户界面上是可见的。例如，在一个借贷系统中，借款的基本条件可能会提供给最终用户（例如，借款人必须年满18岁）。类似的，系统管理员可能要求能够更改业务规则，这需要一个访问这些业务规则的界面以允许执行这种修改。
- **业务规则使用的数据在哪里？**答案指出了根据规则需要输入的数据，处理业务规则的组件需要安置在哪里。这个考虑因素在分布式系统中尤其重要，其中网络延迟对系统的性能有重要的影响。
- **业务规则有可能影响系统性能吗？**有时业务规则具有对系统性能隐藏的或意想不到的影响。在放置规则的时候，您应该考虑这些影响。例如，一个判断个人信誉的规则可能会查询这个人过去两年的信用记录。一个条例的变更可能导致需要查询过去10年的记录，系统的性能突然受到影响，因为信誉系统要多花5秒钟完成查询。
- **存在用于管理业务规则的可用的或得到授权的中间件吗？**如果您正在打算让一个规则引擎成为系统的一部分，您可以用一个分离的组件来表示这个引擎（当物理架构定义后，您将选取这个引擎）。一个规则引擎通常提供定义、管理和执行业务规则的能力，而且几乎肯定会要求这些规则放置在哪里和如何放置。

在组件中放置业务规则的两个关键目标之一是，尽可能地把业务逻辑放置在一个地方（以便于维护）并防止它们在系统的不同部分重复。由于系统的其他特征，并不一定能做到这一点。例如在一个客户端—服务器环境中，业务逻辑可能分布在客户端（如本地数据验证）和服务器端（如业务处理逻辑）。

从描述业务规则的角度来看，在逻辑架构中您有好几个选择：

- 业务规则可能以算法的形式来指定，用以计算结果，或以某种方式操作或检查数据。验证信用卡卡号的规则将应用信用卡机构提供的标准算法来检查一个信用卡的格式是否正确。
- 业务规则可能以组件之间关系的形式来指定，例如“订单必须有一件或多件商品”。
- 业务规则可能以组件管理的数据中的不变量或约束的形式来指定，例如“订单号的字符长度必须是8位”。
- 业务规则可能以运算操作的先决条件或后置条件的形式来指定，例如“每笔订单的金额必须小于200美金”。

《Business Rules Applied》（Von Halle 2001）讨论了专用于业务规则的原则。这些原则被称为STEP规则（Separation、Trace、Externalize和Position）。它们指的是把业务规则从组件中分离、业务规则具体化、追溯业务规则来源的能力和定位规则的变更。

架构决策

架构决策（编写在**架构决策**工作产品中）记录了已经作出的重要决定，也指出架构在结构和关系中的关键方面。架构决策可能会确认形成架构所必须的部分产品或库，从而影响确认的组件。例如，也许您已经决定使用一个由外部供应商提供的特定的安全组件。您还应该确认任何可能已经使用的可重用资源，正如我们在**调查架构资源**任务中讨论的。

对于逻辑架构中在哪里和如何描述组件存在着混淆。尽管记录一个现有元素（物理层面）的使用事实很重要，但是，别搞混了在同一模型中的逻辑和物理层面的元素也同样重要。对于这种情况，推荐的方法是抽象物理层面的元素并以一个逻辑的形式描述，同时对已经决定了实现细节的组件做一个注释。

假设已经决定使用 Acme 公司的 Message Broke 作为消息总线，为了在逻辑功能模型中描述这个决定，您可以把它抽象为一个名为消息总线的组件，并在注释中标明这个组件的实现将使用 Acme 公司的 Message Broke。

这个消息总线组件和前面已经说明过的其他组件一起都列在了图 8.15 中，连同组成 YourTour 架构的层（layer）。我们根据《Software Reuse：Architecture，Process and Organization for Business Success》（Jacobson 1997）一书给出的分层方法和后面的“概念：层级与层”部分中讨论的方法得到了这个系统分层。如果需要，您也可以在一个类似的图中显示包含这些组件的子系统。

任务：概述部署元素

目　　的

这项任务的目的是确认所开发的系统未来部署的地点和在这些地点内的节点。

角　　色

应用架构师（次要的）、数据架构师（次要的）、基础设施架构师

输　　入

架构决策（可选）、架构概览、架构概念证明（可选）、企业架构规范、现有 IT 环境（可选）、功能模型、功能性需求、术语表、非功能性需求、系统上下文

输　　出

部署模型

步　　骤

- 确认地点。
- 确认节点。

架构师角色

架构师负责这项任务。

任何中等复杂的 IT 系统架构通常都由好几个完全不同的、通常分布于一个或多个地点的平台（最终用户桌面、中型服务器、web 和应用服务器、数据库服务器等）组成。这些地点可

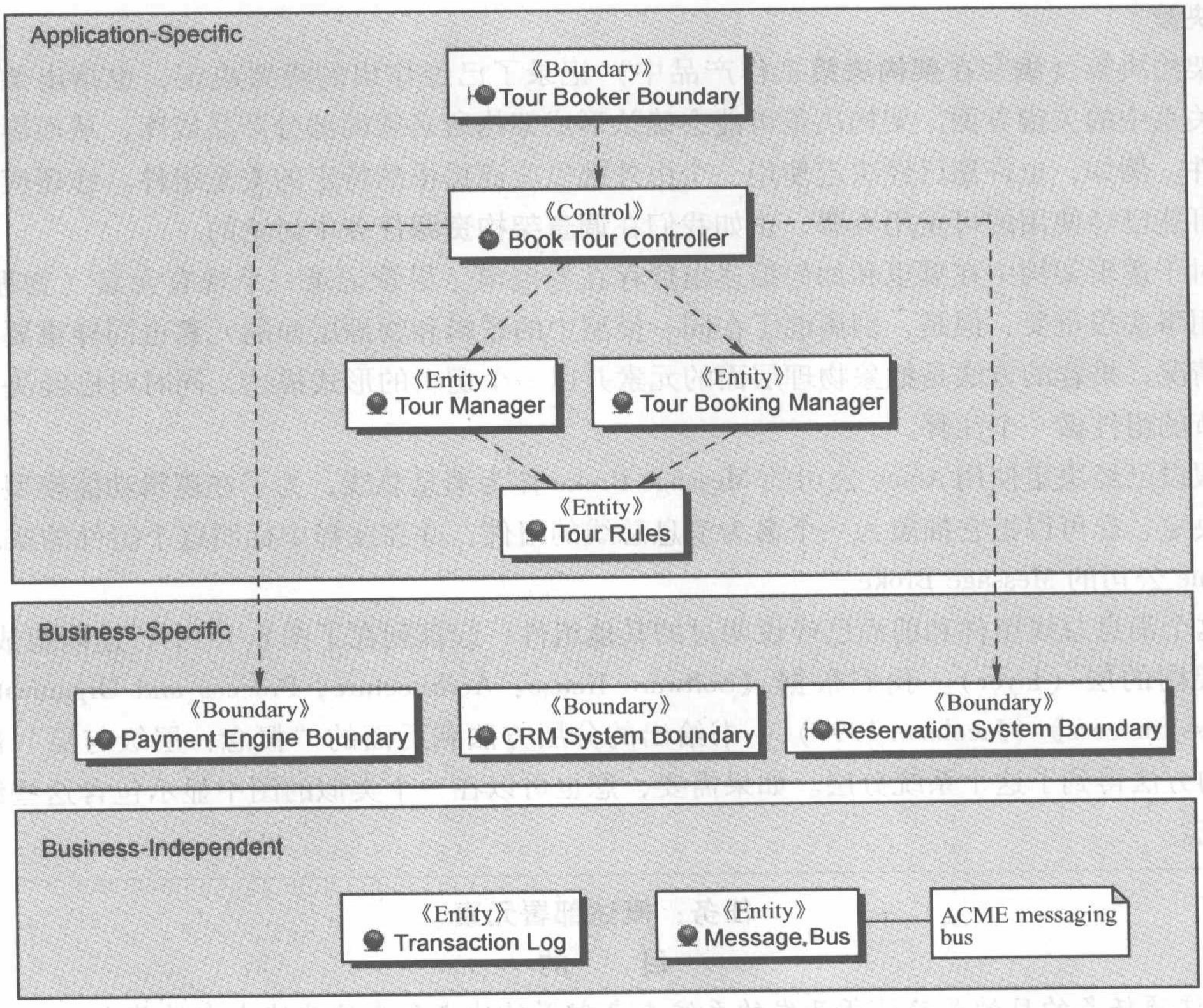


图 8.15 YourTour 组件和层

能仅仅是同一建筑物中的不同房间或楼层，或位于同一地点的不同建筑物，或者它们的位置遍布国内甚至全世界。接下来的挑战是决定它们需要使用什么计算机和平台，以及把它们放在什么地点。这项任务的目的是确认计算机平台分布的地点范围和放置组件的计算机平台本身。

在本书中，我们采用了把软件和硬件视为同等重要的系统工程原理，正如补充内容“概念：软件工程和系统工程”所述。

在这项任务中，您正在确认逻辑地点和节点。您可以使用“一个分支机构”，当您在后面考虑物理架构时或许会把它考虑为“纽约分公司”或“东京分公司”。

最初的一组部署元素可能从**架构概览**中得到。如同功能性元素一样，根据术语表中适当的术语对部署元素命名。

概念：软件工程和系统工程

正如我们在第 2 章中略微谈到的，系统工程领域把系统看作为由软件、硬件、信息和工人组成。当推导一个解决方案时，系统工程师会对使用的软件、硬件和人进行权衡。如

果性能重要，那就可能会作出在硬件中实现某些系统元素的决策，而非软件或人。另一个例子是为客户提供一个可用的系统，所作的一个决定是安排一个人作为和客户的接口，而不是在软件或硬件中实现的接口。更复杂的情形可能需要通过软件、硬件、人的组合达到某种系统质量。因此，本书在适当的地方指的是这些元素，而并不仅是软件。

有趣并值得注意的是，系统工程明确地考虑把软件和硬件（还有人）同等看待。这样可以避免两种缺陷：（1）硬件被看作是二等公民和承载软件的工具；（2）软件被看作二等公民和帮助硬件按期望的那样运行的工具。

概述部署元素任务使用的这种方法和**概述功能性元素**任务中的方法类似，在执行任务时，使用独立于技术的方式考虑架构。在这个阶段，您不知道一个分支机构是否会实现为一个商店，或一个特定国家的网站，或是一个电话联系人等。当考虑逻辑架构时，您所知道的就是一组需要提供的功能。

系统工程领域会非常广义地看待如何从物理上实现一个地点。地点可以包含软件、硬件、人和信息的任意组合。但是，在本书中关注软件密集型系统。正如您将在所举的样例中看到的，本书假定一个地点最终会实现为一个物理的地理位置，它包含物理的计算资源和在这上面部署的软件。

在系统工程领域，系统除了包含软件外，还包含硬件和人，我们都知道某些质量（如可扩展性）需要通过软件和硬件的组合才能达到。因此，当处理这些需求时，您应该同时考虑这两个方面。然而在实际中，**概述功能性元素**任务和**概述部署元素**任务是单独考虑的，以便架构师能够关注特定于架构这两个方面的方法。这样做的风险是忘记考虑软件和硬件是同一个问题的两个方面。因此，我们专门引入**验证架构**任务以确保功能性元素和部署元素协调一致，并确保它们按照一致的方式处理需求（包括功能性的和非功能性的）。

步骤：确认地点

在这个步骤中，您会复审相关的工作产品以确认所开发系统将要分布的地点范围。地点的确认是建立在对下列工作产品的分析之上的：

- **架构决策**工作产品可能建议在特定的地点使用额外的可重用资源（除了在**现有 IT 环境**中已经确认的）。（例如，一个已经作出的决策是要求和位于一个特定城市的主机实现接口）。
- 仅当**企业架构规范**工作产品使用或创建特定的地点时，才可能存在可以遵循的规范。一个关于业务连续性的规范可能会声明：不管出现系统中断或意外的（灾难）事故，系统必须保持运行。这当然意味着使用分开的灾难恢复地点。
- **现有 IT 环境**工作产品可能包含某些您会选择纳入架构的元素，这些元素可能暗示了特定的地点。
- **非功能性需求**工作产品可能确认某些约束，而这些约束为解决方案的某些方面规定了特定的地点。（例如，“呼叫中心必须位于我们的总公司”）。另外，系统必须体现出来

的某些质量（如可靠性）也可能暗示了地点信息，如主要和次要的灾难恢复地点。

- **系统上下文**工作产品确认了所有参与者的地点，无论这些参与者是人或外部系统。

图 8.16 中的拓扑关系显示了考虑 YourTour 系统各种输入信息后获得的最初确认地点。图中显示了一个远程办公室（Remote Office）地点，在那里客户和旅游组织者可以通过互联网接入设备访问 YourTour 系统；一个分公司（Branch Office），在那里顾客和旅游组织者可以通过联系销售员预定旅游线路；一个总公司（Central Office），在那里所有远程办公室和分公司的信息都综合起来；一个 MyPay 数据中心（Data Center），MyPay 系统安置在那里；一个 MyReservation 数据中心，MyReservation 系统安置在那里。尽管 Mypay 数据中心和 MyReservation 数据中心不在您的直接控制内，但是，您仍可能要把某些 YourTour 解决方案的元素部署在这些地方以便和相关的系统集成。这就是为什么要包括这些地点。当您开发详细的集成解决方案时，这些信息将获得确认（或否定）。

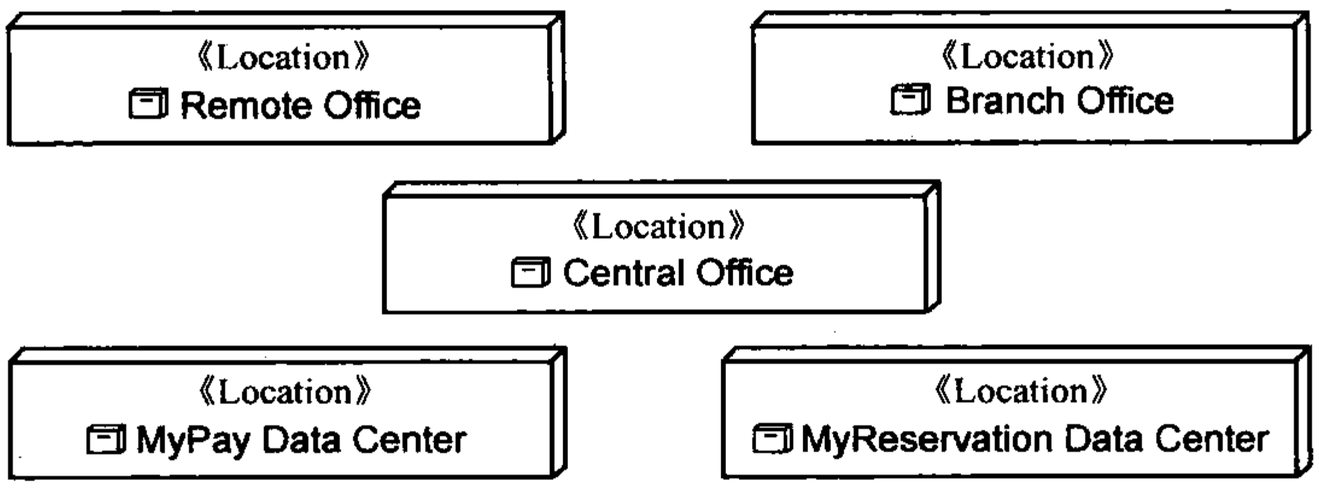


图 8.16 YourTour 系统中的地点

步骤：确认节点

在确认地点以后，您将继续确认位于这些地点内的节点以及组件最终部署在哪些节点上。在这个阶段，节点纯粹是逻辑性的容器，没有确定任何技术相关的或特定产品的属性（除非已经由特定的约束标明）。您会在考虑物理架构时关心这些细节。

在这个步骤中，架构师复审所有能够得到的输入来确认所开发系统分布的节点范围。节点的确认基于对工作产品的分析，特别是对**功能性需求**、**非功能性需求**、**功能模型**、**现有 IT 环境**和**架构决策**的分析。我们会在接下来的部分讨论这些工作产品。另外，您还可能考虑任何已经创建的**架构概念证明**。

功能性需求

功能性需求工作产品会帮助您理解谁是系统的参与者以及每个参与者需要的功能。根据这些信息，您可以决定需要什么计算资源来支持系统的运行。

非功能性需求

非功能性需求工作产品可能会提议系统需要具有某些特定的服务质量或一个特定质量的不同服务级别，然后这个提议会意味着需要引入不同的节点。为得到每个节点必须体现出的质量，可以按照类似于处理组件的方法，在这个方法中您需要依次考虑每个非功能性需求并最终把需求分配给这些节点。

现有 IT 环境

现有 IT 环境工作产品确认已经存在的节点以及哪些可能用来支持所开发系统组件的运行。包括将要构建的新组件，或作为解决方案的一部分已经存在并需要实现接口的组件。

功能模型

功能模型工作产品确认部署环境需要支持的组件，这可能建议需要特定的计算资源。

架构决策

架构决策工作产品可能建议使用特定的节点。以 YourTour 系统为例，图 8.17 显示了安置在总公司的节点。这张图基于下面的假定：当前所有和旅游线路有关的组件（既包括旅游线路处理的组件也包括旅游线路服务的组件）将部署在一台逻辑服务器上——旅游线路预定服务器。以后在您基于对需求的理解和所有需求（特别是那些和质量相关的）的满足程度以及已经创建的任何架构概念证明的执行结果而精炼架构时，您可以改变这个决定。

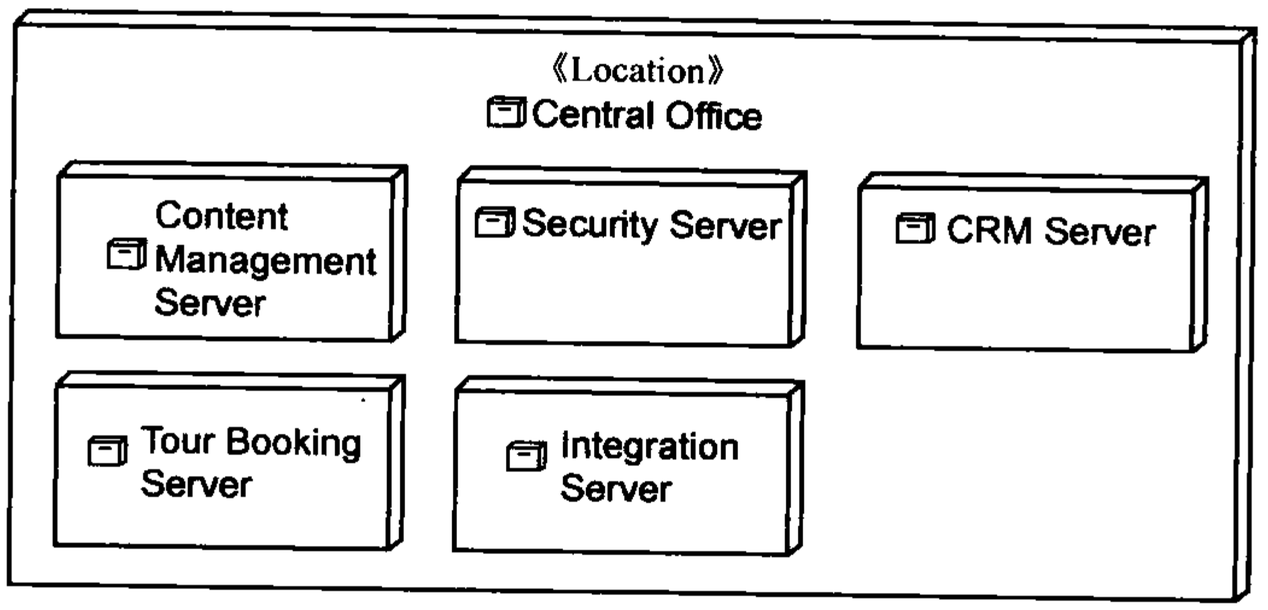


图 8.17　分配给总公司的节点

另外，内容管理服务器将装载一个内容管理应用程序。由于安全的原因，也是因为内容开发具有不同的可靠性需求，一台单独的服务器将用于内容管理。尤其是内容只需要在正常的工作时间内开发，而旅游线路预定服务器（Tour Booking Server）则有严格得多的可靠性需求。

安全服务器包含专用的与安全相关的软件，而且您预计这个功能需要和其他节点的组件在物理上分离。类似的，还提供一个单独的集成服务器来接受任何必需的特定集成软件。

任务：检验架构

目　　的

这项任务的目的是检验所有的架构产品是否一致，并确保跨越架构工作产品的所有问题都得到一致的解决。

角　　色

应用架构师（次要的）、数据架构师（次要的）、基础设施架构师（次要的）、首席架构师

输　入

架构决策、架构概览、架构概念证明、部署模型、功能模型、功能性需求、非功能性需求、系统上下文

输　出

复审记录

步　骤

- 计划检验。
- 召开动员会议。
- 进行个别检验。
- 召开检验会议。
- 进行返工。
- 进行后续工作。

架构师角色

- 领导对架构的检验以确保全部架构工作产品一致。
- 确保跨越架构工作产品的问题（质量，如性能）得到一致的解决。
- 确保解决检验中发现的缺陷且不破坏整体的架构。
- 确保记录任何检验活动中得到的架构决策。

很多原因会导致架构工作产品不一致，例如实际上有时候它们由不同的角色构建。比如，**功能模型**由**系统架构师**构建，而**部署模型**由**基础设施架构师**构建。尽管您可能觉得已经在**功能模型**中解决了某些和外部系统集成的问题，但是，**部署模型**中所作的一些决策会使这些假定失效。

因此，这项任务的目的是依次查看每个工作产品以确保在您继续提供更多的细节之前任何相关的工作产品之间保持一致性。正如在**细化功能元素**和**细化部署元素**任务中所述。这项任务应该确保任何跨越架构工作产品的问题得到一致的解决。在检查这种一致性时，您也要参考所有的需求工作产品，特别是**系统上下文**、**功能性需求**和**非功能性需求**工作产品。检验（Verification）和确认（validation）并不相同，确认关注系统的整体而非个别工作产品。请看后面的“概念：检验与确认”部分对这两种技术的讨论。

概念：检验（Verification）与确认（Validation）

在本书中，我们使用了软件工程研究所（SEI）建立软件成熟度模型（Ahern 2008）的方法中定义的术语——检验和确认。

检验允许架构师判断在整个开发进程中，当前的工作产品是否和为创建它们而使用的输入工作产品一致（例如其他工作产品和任何强制的开发标准）。广义地来看，检验帮助您及早地察觉到工作产品中的缺陷，因此避免后期返工。检验帮助您回答“我正在正确地

构建系统吗?”用于支持检验的技术包括检验工作产品、从解决方案的组件追溯到需求和使用自动检验的工具，一些建模环境中提供了这样的工具。

确认关注整个系统而不是个别的工作产品，它关系到确保系统满足声明的需求。确认帮助您回答“我正在构建正确的系统吗?”换句话说，系统满足声明的需求吗？或构建了其他的一些东西吗？这个过程既考虑功能性需求也考虑非功能性需求，还包括一些重要的时间、预算、资源约束。确认通常由相关主题的专家和利益相关者来复审，还有黑盒测试。

检验的目标之一是发现并去除出现在工作产品中的缺陷。有许多种技术可以用来达到这个目的，如复审（review）、普查（walk-through）和检查（inspection）。这项任务的正式程度取决于许多因素，尤其是架构的复杂性。在一个极端情况下，这项任务可能是较为简单的非正式复审，需要架构师查看架构工作产品，留意任何不一致的地方，然后相应地纠正这些工作产品。对于另一个极端情况，您使用严格的技术，依次对所有的架构工作产品进行系统的普查。我们在这里阐述的就是这种方法。我们的假设是这里阐述的方法可以根据情况容易地变化并根据需要进行缩减。

这项技术基于检查的概念并涉及根据相关工作产品（如作为输入的那些工作产品）查看一个指定工作产品。另外，一个检查列表可以用于帮助复审人员。例如在检查**部署模型**时，复审人员可以使用检查列表中的各项来确保所有**功能模型**中的组件全部分配给**部署模型**中的各个节点。

这项任务中的步骤遵照 Michael Fagan（Fagan 1976）最先提出的经典的检查流程。图 8.18 显示了这个流程，为了适于检查架构工作产品，它已经进行了适当地调整。如果想要更全面地了解用于软件开发的检查流程，请参考《Software Inspection》（Gilb 1993）。

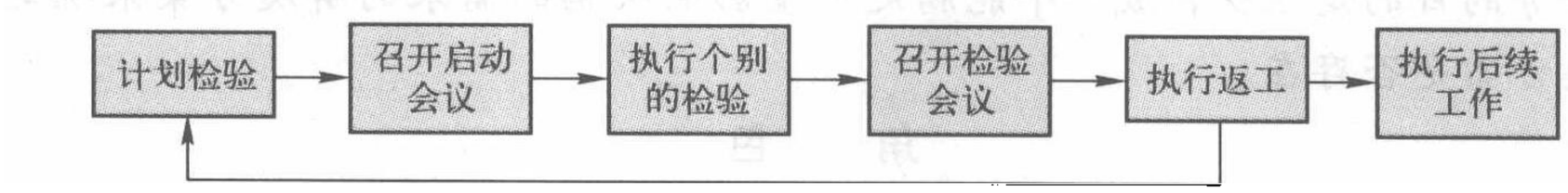


图 8.18　检验步骤

步骤：计划检验

全面的计划是架构检验成功的关键。所有需要检验的工作产品都集中起来，所有的参与者都确定下来，还要决定如何通知参与者（例如通过 e-mail 或启动会议）。

步骤：召开启动会议

这个步骤是可选的，但是如果参与者不熟悉检验的概念、他们在检验中的角色，或检验的对象，则需要召开启动会议。会议的目的是给团队成员提供一些关于检验材料的背景知识。可以给参与者提供一些检查列表来帮助集中他们的评论，因为每一个参与者都被要求查看相关工作产品中的特定方面。

步骤：执行个别的检验

参与者得到了他们需要核查的材料，然后花时间检查这些工作产品并对他们发现的缺陷进

行标注。清晰地完成这项工作所需的时间依赖于所检查工作产品的大小。尽管这个活动看起来很繁重，但是，及早找到缺陷可以减少返工。为了协助工作产品的检查，可以给参与者一个确定哪些特定的东西需要检查的检查列表。

步骤：召开检验会议

这个会议的目的是正式地记录并归类所有在个别检验中确认的缺陷，还包括其他的难点或问题。召开一个单独的会议来检验架构几乎总会导致更多的缺陷被参加的人发现，这些缺陷在他们单独检查的时候没有找到。检验会议不是解决问题和讨论分歧的地方，它只是一个关注**复审记录**中记录的缺陷的途径。为了使检验会议顺利地进行，需要一个有经验的主持人主持，他能够迅速地停止争论，当技术人员集中在一起的时候很可能会发生这种情况。

步骤：进行返工

返工是检验任务的检验会议中发现的缺陷由作者解决的一个步骤。相关的工作产品需要进行相应的更新。理解导致缺陷的原因和流程中的什么地方最容易产生缺陷也是一种智慧。最终，开发流程得到改进。如果返工十分广泛，请返回“计划检验”重新进行这个流程。

步骤：进行后续工作

进行后续工作这个步骤确保在检验会议上发现的所有缺陷都得到改正。实际上是协调人负责这个步骤。

任务：构建架构的概念证明

目　　的

这项任务的目的是至少合成一个能满足严重影响架构的需求的解决方案来确定架构师设想的解决方案是否存在。

角　　色

应用架构师（次要的）、数据架构师（次要的）、基础设施架构师（次要的）、首席架构师

输　　入

架构决策、架构概览、部署模型、功能模型、功能性需求、RAID 日志、非功能性需求、排定优先级的需求列表

输　　出

架构概念证明

步　　骤

- 创建架构的概念证明。
- 把发现的内容编写成文档。

架构师角色

- 发起概念证明并确立基本规则和概念证明的范围。

- 和所有的产品供应商紧密合作以对于概念证明需要他们提供什么产品给出明确的指导。
- 比较和解释概念证明的结果并向利益相关者汇报。
- 根据概念证明的结果，作出任何需要的架构决策。

建立架构概念证明的目的是减小风险。这项工作产品可采用多种形式，从一个似乎适合解决方案的已知技术列表或一个简单的解决方案概念模型草图到一个可运行的原型系统。

对于那些特征不能在书面上（或模型中）进行决定的架构以及那些需要一些经验性的证据来证明架构师的假设成立（或不成立）的架构，架构概念证明将采用可执行软件的形式。一个需要采用可执行的原型程序的例子是：当您需要判断系统的点对点的性能时，虽然可以在纸上通过理论计算得到，但是，最好的判断来自解决方案所设想的典型环境中运行的程序。构建一个概念证明可能需要可观的工作量和成本，因此只有认为风险大到不减轻就可能导致以后花费更大的成本时才构建概念证明。

构建和运行架构概念证明的结果需要编写成文档并用于检验假设，或当存在多种选择时决定哪些选项比其他的更切实可行。

步骤：创建架构概念证明

有很多原因导致需要创建一个架构概念证明，其中包括下面这些原因：

- 功能性需求新颖或具有挑战性。例如，您可能正在把新的业务功能或流程引入到组织中。
- 所开发系统有特殊的质量要求，例如高可靠性或极快的响应时间。
- 需要使用新的和没有经过验证的技术（至少在开发团队看起来是这样），例如新的现成的商业（COTS）程序库。
- 将采用新的标准。
- 和现有系统的接口很复杂或没有文档。
- 将使用新的开发工具，您需要证明项目能够随着新的工具继续进行。

对于这项任务的输入工作产品反映了这些不同的考虑，因此需要包括**功能性需求**、**非功能性需求**和**排定优先级的需求列表**这些工作产品，还包括**架构概览**、**部署模型**和**功能模型**这些架构工作产品。另外，上述的一个或多个关注点可能已经确认并作为风险或潜在的难点记录在**RAID 日志**中。除此之外，也许在**架构决策**工作产品中记录的一个或多个架构决策无法得到满足，直到一些原型工作完成来测试各种选项。

在 YourTour 系统中，一些关于和外部的 MyPay 支付引擎及 MyReservation 预定系统集成的在访问机制和性能方面的特定挑战已经确认。您还关注在这些集成中使用的安全模型。您需要创建一个架构概念证明来考虑架构的这些方面。

整理一个架构概念证明可能很有挑战性，因为根据定义，这项工作和制定架构的工作同时进行，并在一定程度上不断变化。还要记住，概念证明通常看作是一次性的，不允许忽视可能

导致的内在危险而演化成真实的系统，这就像构建一个健壮性和可扩展性不足以支持全部需求的基础。

步骤：把发现的内容编写成文档

执行概念证明的结果被编写成相应的文档，作为**架构概念证明**工作产品的一部分。

任务：细化功能元素

目　　的

这项任务的目的是把功能性元素精炼到可以交付给详细设计的程度。

角　　色

应用架构师、数据架构师（次要的）、基础设施架构师（次要的）

输　　出

架构决策、业务实体模型、功能模型、术语表

输　　入

数据模型、功能模型

步　　骤

- 定义组件接口。
- 定义操作和操作签名。
- 定义组件之间的契约。

架构师角色

架构师负责这项任务。

我们已经在**概述功能性元素**任务中概括了功能元素，在这项任务中要对这些元素充实一些细节。具体地说，这项任务包括：

- 定义您的组件提供的和要求的接口。提供的接口是指组件提供给其他组件使用的接口。要求的接口是指组件必须使用来完成某些职责的接口。
- 说明接口的操作和它们的签名（也就是传入的参数、参数类型和返回值）。
- 从先决条件和后置条件的角度说明每个操作行为的规约。

您也许会问“我为什么要关心得这么详细，尤其是在一个逻辑架构中？”我们认为有下列两个原因：

- 如果架构师没有把组件说明得如此详细，那就留给设计者和实现者完成这项任务。他们的工作通常不受架构师的直接影响（甚至可能没有，即使在同一个公司或国家），因此，他们有可能作出和架构不一致的决定。
- 以这种方式说明组件允许您在逻辑功能模型和逻辑数据模型之间进行连接。这些模型通常单独或独立地开发，这个方法允许您在项目的早期确保两个模型保持一致。

尽管如此，组件可能并不总是需要定义到如此详细的程度。例如，预计使用一个产品或库，或其他现成的组件来实现一个组件，这样就不需要说明的像这里讨论的那么详细。

这项任务认可在**架构决策**工作产品中记录的任何决定，也根据**术语表**中的术语命名功能元素。

步骤：定义组件接口

组件通过接口来交互。接口是一个组件访问另一个组件的服务以及同另一个组件交换数据的媒介。采用这种方式是一个很好的做法，因为使用者仅依赖于接口而不是接口提供者的实现。以后用另一个实现来替换这个实现会很容易，只要遵循相同的接口。

组件的特征在于它提供的接口（和通过每个接口可用的操作）和它要求的接口。在实现一个组件提供的接口时，这个组件可能需要其他组件的服务，在这种情况下，它们使用其他组件提供的接口。正如您将看到的，组件提供的和要求的接口源于分配给组件的职责。

接口和实现的分离还允许您在从逻辑架构转向物理架构时作出更明智的选择。接口的职责（由接口后面的组件实现）分离得越清晰，您就越能够有效地选择对应的物理架构元素。处理接口（正如我们在后面的“概念：先决条件和后置条件”部分描述的，操作、先决条件和后置条件）意味着您已经拥有一个非常精确的说明，允许您更精确地设计物理组件或找到更适合这个接口规范的产品或库。

在这个步骤中，您把接口分配给先前确定的各个组件。图 8.19 中显示了一个组件提供的接口的例子：旅游线路预定管理器（Tour Booking Manager）组件提供了接口 ITourBooking 和 IPayment。

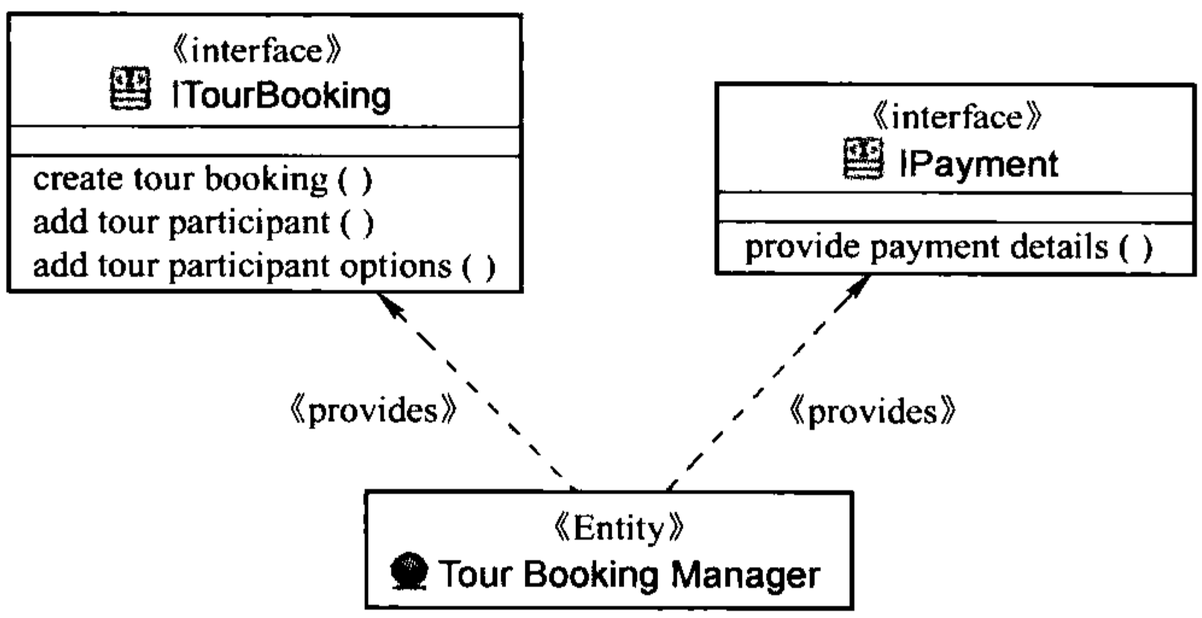


图 8.19　一个组件提供的接口

图 8.19 中的接口来自哪里？您需要考虑之前分配给组件的职责并根据它们的相似程度（或者耦合程度）进行分组。在这个例子中，所有分配给旅游线路预定管理器（Tour Booking Manager）组件的职责都集中在 ITourBooking 接口中，负责处理支付的职责单独地放在 IPayment 接口中。按照惯例，接口的名称都加上前缀 I。接口的分配依次应用到所有的组件。

您可能会选择精炼所有的需求实现来显示如何使用这些接口，而不是提供这些接口的组件。图 8.20 中提供了一个经过精炼的时序图，它显示了预定旅游线路（Book Tour）用例的主要事件流的实现，其中使用了已经确认的接口（与提供这些接口的组件相对）。

与图 8.20 类似的图让您对一个特定组件所要求的接口有更好的理解。结果是您能够显示一个特定组件提供和要求的接口。图 8.21 显示了预定旅游线路控制器（Book Tour Controller）

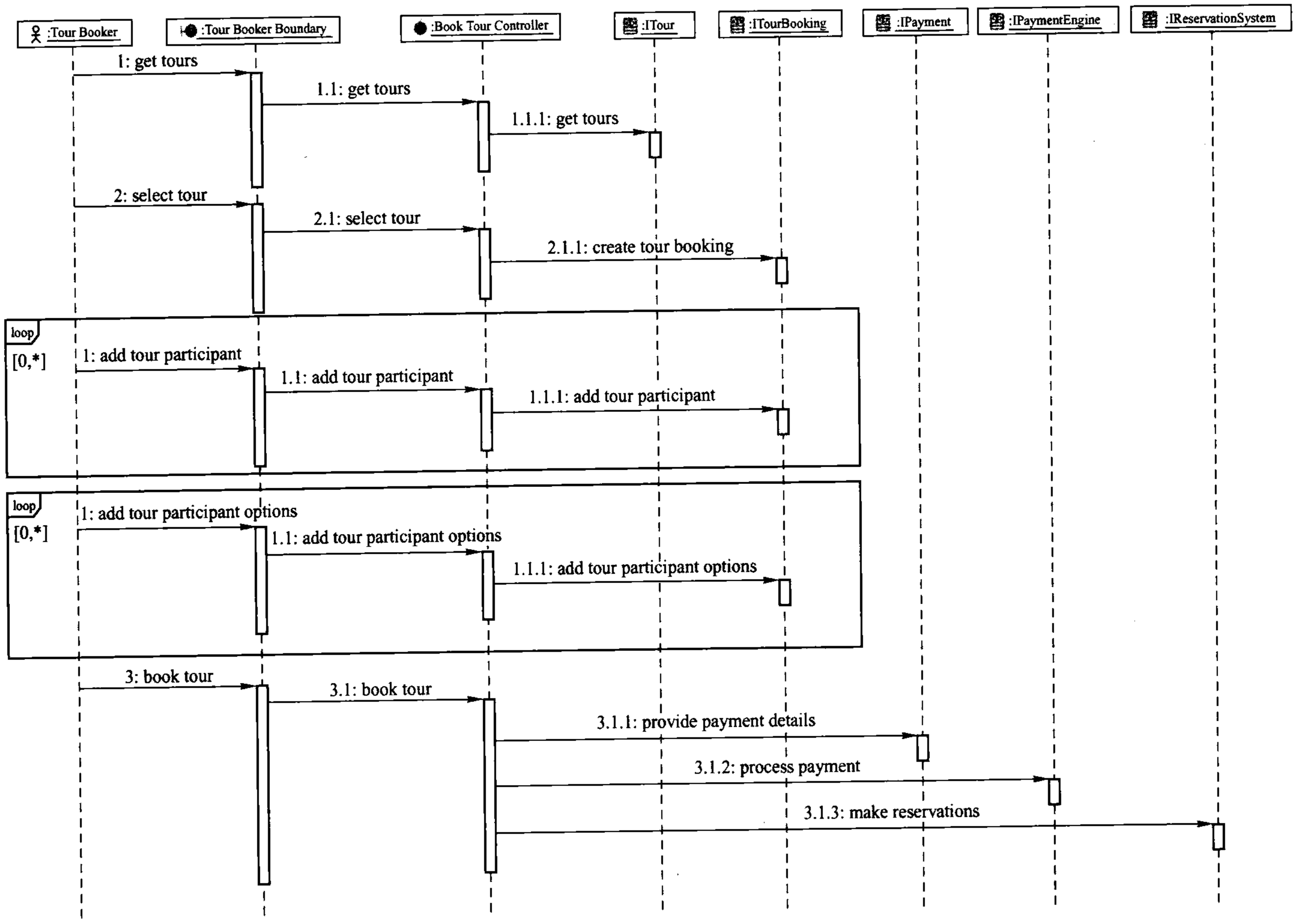


图8.20　预定旅游线路用例（主流程）的UML时序图

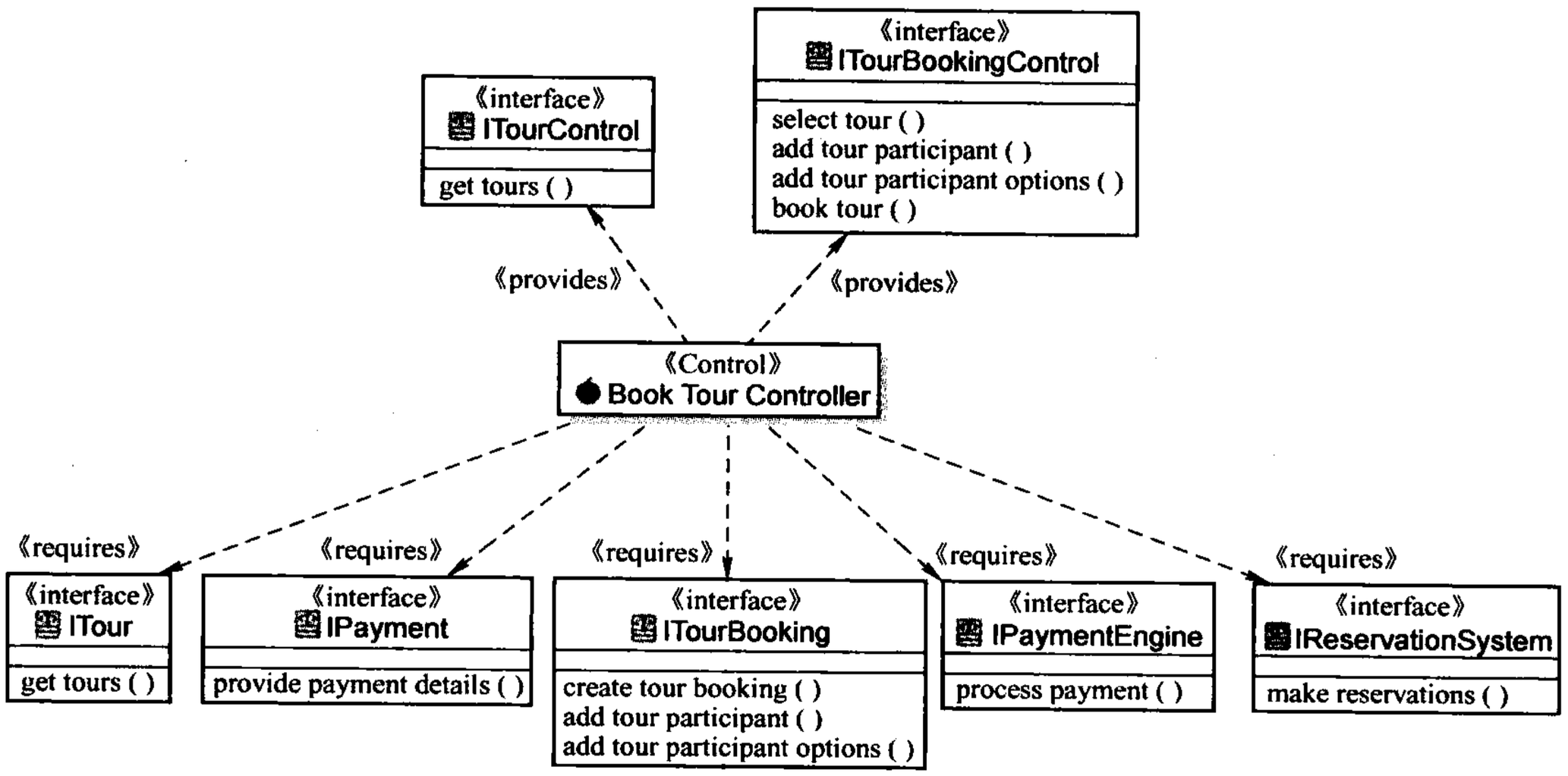


图 8.21　预定旅游线路控制器的组件说明图

组件提供和要求的接口，所需的接口来自图 8.20 和其他类似的描述每个用例不同事件流的图。这个概图可以提供一个或多个组件详细的说明，正是由于这个原因，这种图通常称为组件说明图（Cheesman 2001）。

正如先前在**概述功能性元素**任务中讨论的，您可以把某些非功能性需求分配给每个组件相关的操作。因此，当定义和一个组件相关的接口时，您可以把这种分配提升到接口上同等的操作。先前，有一个 6 秒钟的性能需求分配给了预定系统边界（Reservation System Boundary）组件中的 make reservations( )操作。这个需求依次分配给 IReservation 接口上同等的操作。

步骤：定义操作和操作签名

在这个步骤中，您将通过指定操作的实际名称和它的签名（Signature）来更详细地定义接口。操作的名称通常和您前面分配给各接口的职责相同，尽管可能会按照需要缩短或简写。

每一个操作（包括传入的和返回的）的签名在某种程度上可以由它的名称决定。您可以假设 add tour participant( )需要接受旅游参与者增加的旅游线路预定信息和旅游参与者的信息。您也可检查之前创建的时序图（如图 8.20 所示）来判断数据流。

操作参数和任意的返回值可以是简单类型，如 String 或 Integer，或更复杂的业务类型。这些业务类型代表一个逻辑**数据模型**，它通常类似且源于**业务实体模型**。图 8.22 显示了一个基于 YourTour 系统**业务实体模型**的逻辑**数据模型**（部分的）。图中应用了 BusinessType 这个 UML 构造型。

一个可以进一步探索的方面是系统中流动的数据的持久性。请考虑下列特性：

- **数据是瞬态的**（transient）。这样的数据仅与方法调用的时间段相关。例如，把两个数相加并返回结果的请求不需要组件在函数调用的生命期之外保存任何与计算相关的信息。然而有时候需要保存结果，例如保险金报价的计算结果。

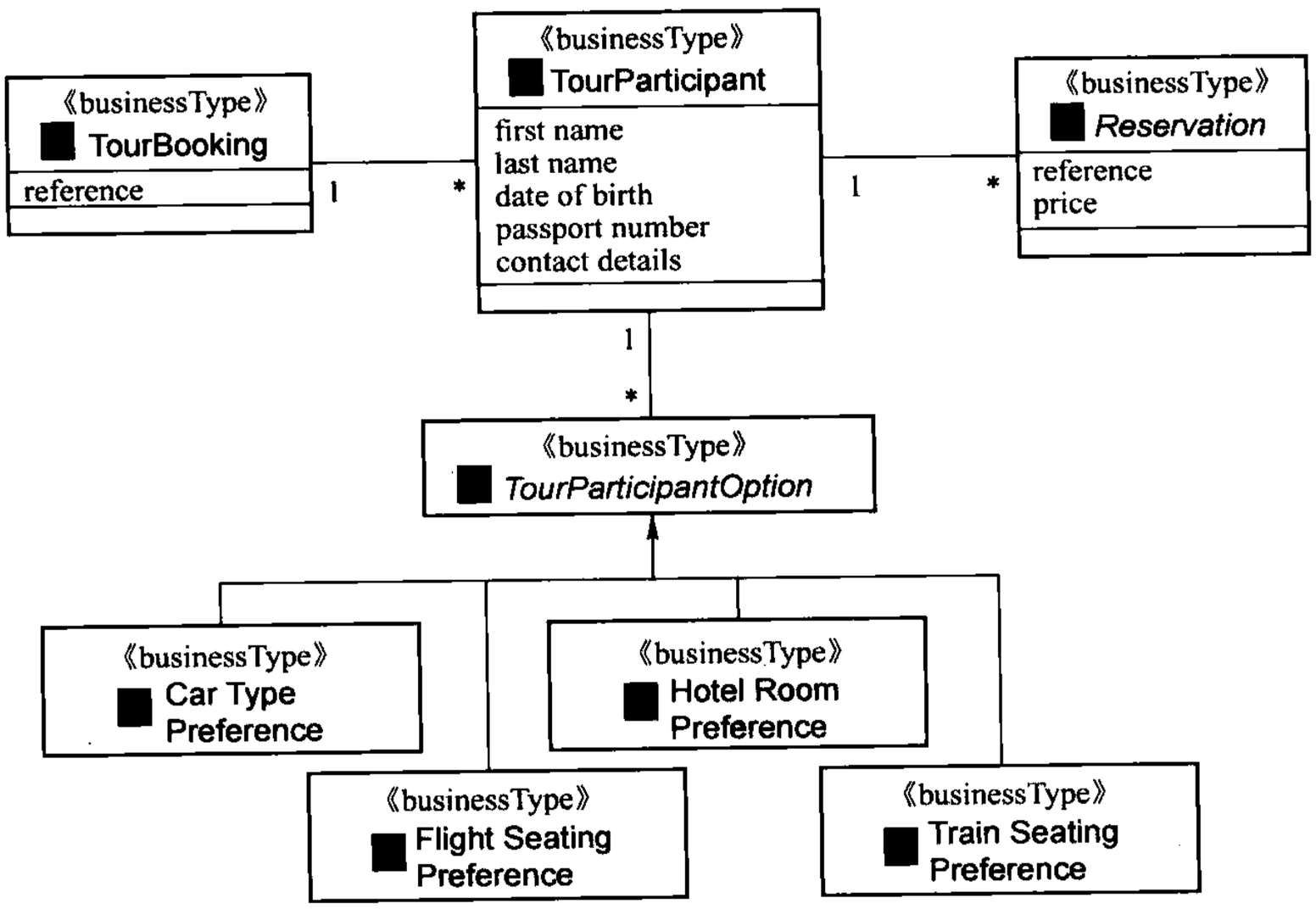


图 8.22 逻辑数据模型

- **数据代表一个会话的状态**。数据仅仅可能在一个用户的会话期间持久。例如，这种数据可能描述购物车里的内容，当用户的会话结束后数据就会丢弃。然而，即使在这种情况下，一个特定的实现也可能在用户的会话之外保存购物车的内容。
- 数据被持久。持久的数据被写入数据存储，这样可以保存到函数的调用生命周期之后、组件的生命周期之后和任意用户的会话生命周期之后。

写入数据存储的持久的数据特别让人感兴趣，因为在逻辑**数据模型**中的业务类型和负责管理这些类型的组件（和接口）之间存在一个关系。Cheesman 和 Daniels（Cheesman 2001）阐述了一种提供这种联系的特殊的技术，在这个技术中作者们引入了接口职责图。这种图的目的是显示哪种接口管理哪种业务类型。接口职责图始终包括接口和它的所有操作（也可以看作是一个组件说明图的接口等价物），以及由这个接口管理的任何业务类型。图 8.23 显示了这样的一个例子。

接口职责图源于逻辑**数据模型**，通过把一个所谓的核心业务类型分配给某个特定的管理那个业务类型（和其他非核心业务类型）的接口。一个核心业务类型实质上是一个不依赖其他类型而存在的类型。在这个例子中，TourBooking 认为是一个核心业务类型，而 TourParticipant 则不是（因为它的存在依赖于 TourBooking）。一个接口通常和单个业务类型相关，并且每个业务类型都只由一个接口管理。ITourBooking 和 TourBooking 之间的关联在图中以 UML 的聚合

（composite aggregation）表示（实心的菱形箭头）。

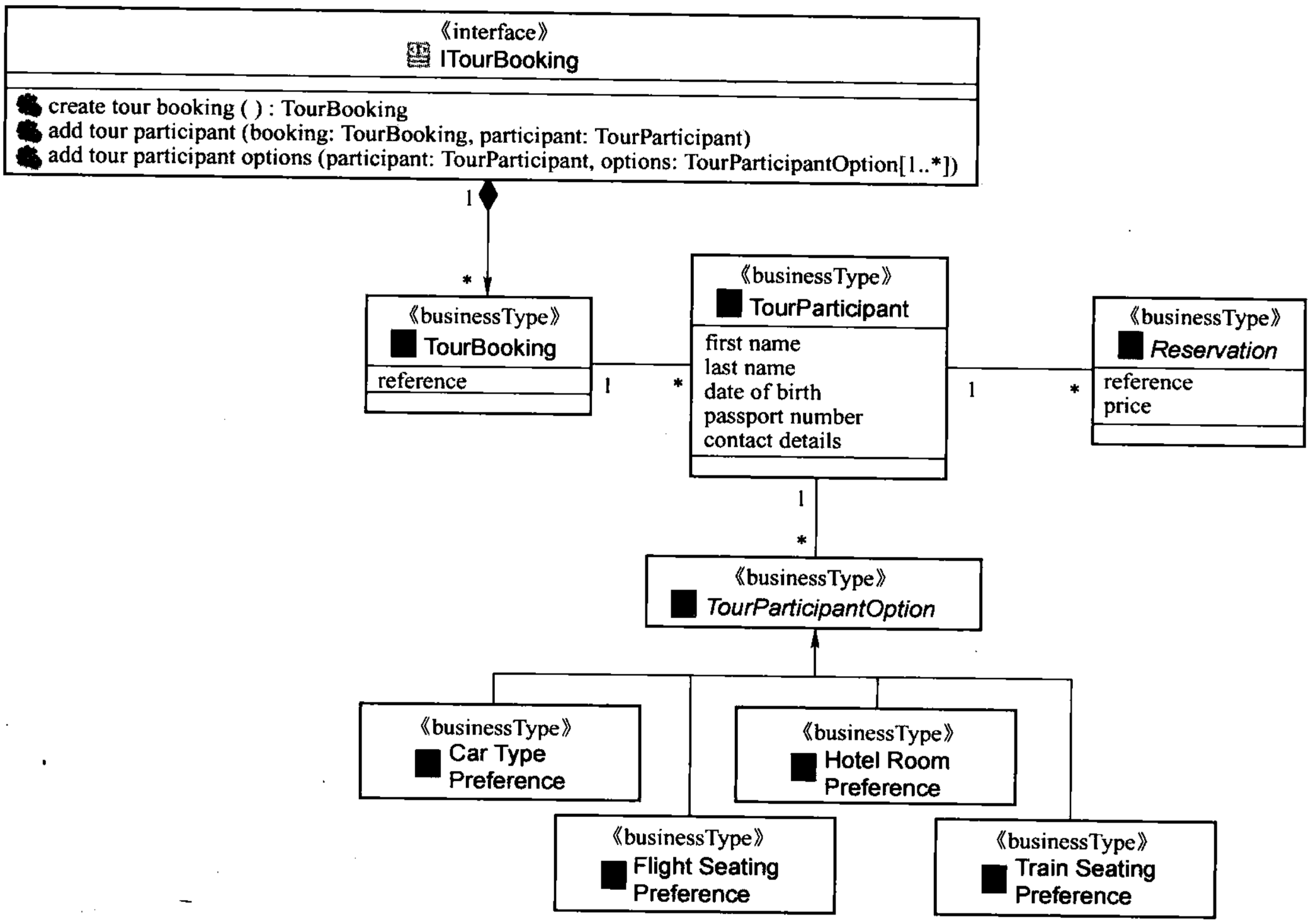


图 8.23　接口职责图

因此，接口职责图是对由接口（和任何提供这个接口的组件）管理的数据的说明。数据本身是持久的，但是，说明中没有提到如何进行持久。就是说，这种说明是逻辑性的，而非物理的。

步骤：定义组件之间的契约

确认了组件和它们的接口之后，您会继续定义这些组件之间的契约。在 20 世纪 90 年代，Bertrand Meyer（Meyer 1997）提出了著名的按照契约设计的概念。从这种意义上来看，这个契约就像两方之间任何合法的契约一样，双方都有履行契约的义务。对于组件，契约以组件提供的接口中的每一个操作的先决条件和后置条件的形式出现。请看“概念：先决条件和后置条件”，它简要地介绍了根据契约进行设计这个概念背后的想法。

概念：先决条件和后置条件

先决条件是在操作执行之前必须满足的一个条件。如果一个先决条件是真，您就可以保证后置条件也是真。然而如果在操作调用时先决条件不是真，操作的结果就得不到保证，

实际上没有指定（就是说，您不能确定操作将做什么和返回什么结果）。

后置条件指定了操作调用之后组件的状态，前提是先决条件是真。我们所指的是组件职责范围之内的任何数据（瞬态的和持久的）的状态。

就像法律一样，先决条件和后置条件规定了客户和契约提供者的义务。从软件的角度来看，客户是调用操作的组件，提供者是实现这个操作的组件，就像如图 8.24 所总结的那样。

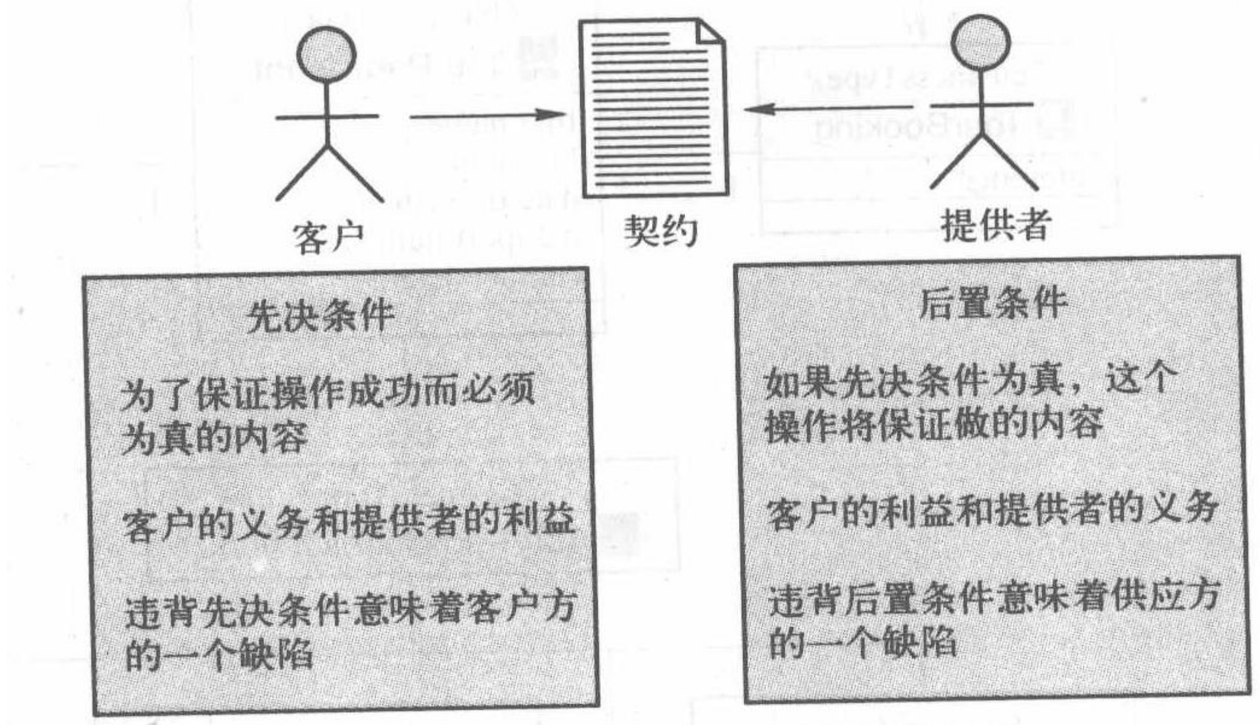


图 8.24 先决条件和后置条件的客户和提供者的义务和利益

组件接口的操作拥有先决条件和后置条件的目的是明确调用者在调用操作之前的职责。换句话说，如果某些事物列入先决条件，提供者不需要检查它们，只需要假设它们是真。这就避免了对传入操作的数据进行过多或过少的检查。在软件中，过多的检查和过少的检查同样有害，因为这样会导致程序膨胀，使软件过于复杂并难于维护，当然，检查错误的代码本身就可能引入更多产生错误的机会！

正如在《Object-Oriented Software Construction》（Meyer 1997）中阐述的，一个先决条件不会停止操作的执行，因此，这个问题就留给该操作的开发人员，由他们决定如何处理（或不处理）无效的先决条件。如果进行处理，那就通常涉及某种形式的异常处理，不管是由运行期环境提供还是由这个操作的开发人员明确地编写。

先决条件和后置条件既可以使用正式的对象约束语言（Object Constraint Language）（OCL）（Warmer 1999），也可以使用自然语言来表达。使用正式语言的一个优点是可以生成支持某种编程语言的强制约束代码（尽管您更有可能根据物理架构产生代码，而不是根据逻辑架构）。

在为旅游线路添加参与者选项中的一个先决条件的例子是“指定的游客选项数量不能是零”，如图 8.25 所示，其中，OCL 已经用来说明这个约束。图中还显示了用自然语言表达的先决条件。

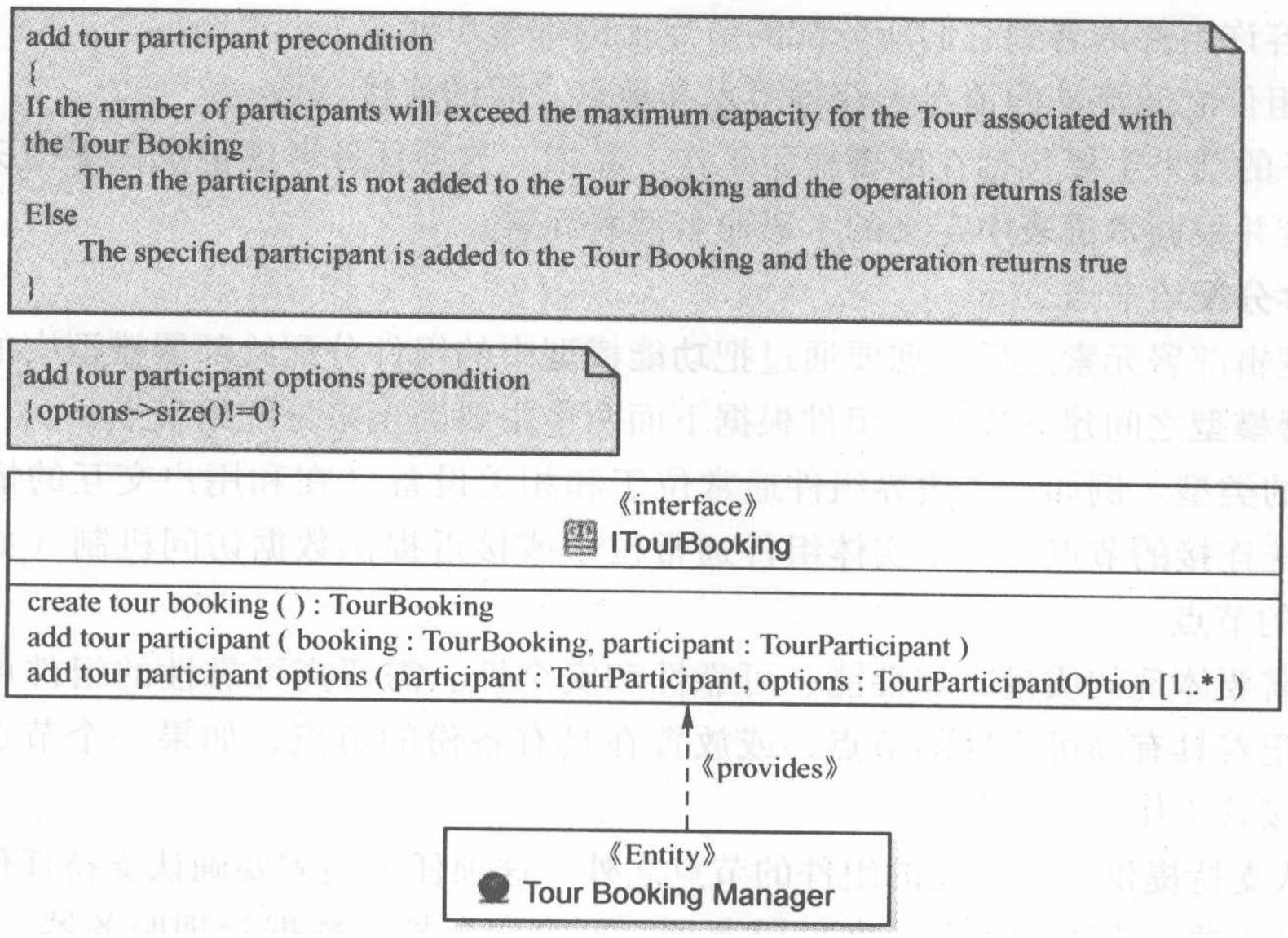


图 8.25　先决条件的例子

任务：细化部署元素

目　　的

这项任务的目的是精炼部署元素以达到可以交付给详细设计的程度。

角　　色

应用架构师（次要的）、数据架构师（次要的）、基础设施架构师

输　　入

架构决策、部署模型、功能模型、术语表、非功能性需求

输　　出

部署模型

步　　骤

- 把组件分配给节点。
- 定义节点之间的连接。
- 定义地点之间的连接。

架构师角色

架构师负责这一任务。

我们已经在**概览部署元素**任务中概括了部署元素，这项任务中您要为这些元素充实一些细节。特别地，这项任务包括：

- 把组件分配给节点。

- 确认容许组件部署到它们所分配的节点上的部署单元。
- 基于组件之间流动的消息来获得节点和地点之间的连接。

这项任务的结果主要包括在**部署模型**工作产品中。这项任务考虑记录在**架构决策**工作产品中的所有决策并根据**术语表**中定义的术语命名部署元素。

步骤：把组件分配给节点

确定了逻辑部署元素之后，您要通过把**功能模型**中的组件分配给**部署模型**中的节点来在**功能模型**和**部署模型**之间建立连接。组件根据下面两个主要因素来分配给节点：

- 组件的类型。例如一个边界组件通常位于和相关设备（在和用户交互的情况中）或外部系统连接的节点。一个实体组件通常位于或接近提供数据访问机制（如数据库服务器）的节点。
- 系统需要体现的质量，如性能、可靠性和安全性。需要高可靠性的组件可能需要放置在标记着具有高可靠性的节点，或放置在具有备份的节点，如果一个节点失败，另一个将接管工作。

除了确认支持提供业务功能的组件的节点之外，这项任务还需要确认支持任何必须服务的节点：打印服务器、应用服务器、文件服务器、安全服务器、数据管理服务器、事务服务器、系统管理服务器等。

图 8.26 中显示了 YourTour 系统中把组件分配给节点的一个例子。这张图介绍了部署单元的概念，它在部署组件和部署节点之间提供连接。我们在补充内容“概念：逻辑部署单元”中对逻辑部署单元进行了讨论。

图 8.26 描述了总公司（Central Office）地点。在这里，您可以看到，在旅游线路预定服务器（Tour Booking Server）中这些元素部署到三个部署单元（一个单元用于每一种类型的组件：边界、控制、实体），一个单独的部署单元用于那些部署到内容管理服务器（Content Management Server）的组件，另一个单独的部署单元用于那些部署到安全服务器（Security Server）上的组件，还有一个部署单元用于那些代表和外部系统的接口的边界组件。一个部署单元由一个 UML 工件（artifact）来表示。

步骤：定义节点之间的连接

当交互的组件放置到各个节点之后，如果它们部署在分布式系统中，您要确保这些组件之间能够相互通信。在逻辑架构中，您不用关心物理通信是端到端（point to point）的还是通过一个中间件的机制，例如一个消息总线；您只需要把这种关联表示为两个节点之间的一根连接线。在考虑物理架构时，您再考虑有关的物理机制。图 8.27 中显示了前面确认的总公司（Central Office）地点中的节点和它们之间的简单关联。

概念：逻辑部署单元

部署单元的概念代表的是一个随后部署到一个或多个节点的组件包。换种说法，部署单元的概念就是容许把组件部署到一个节点。那么您为什么需要这种媒介？

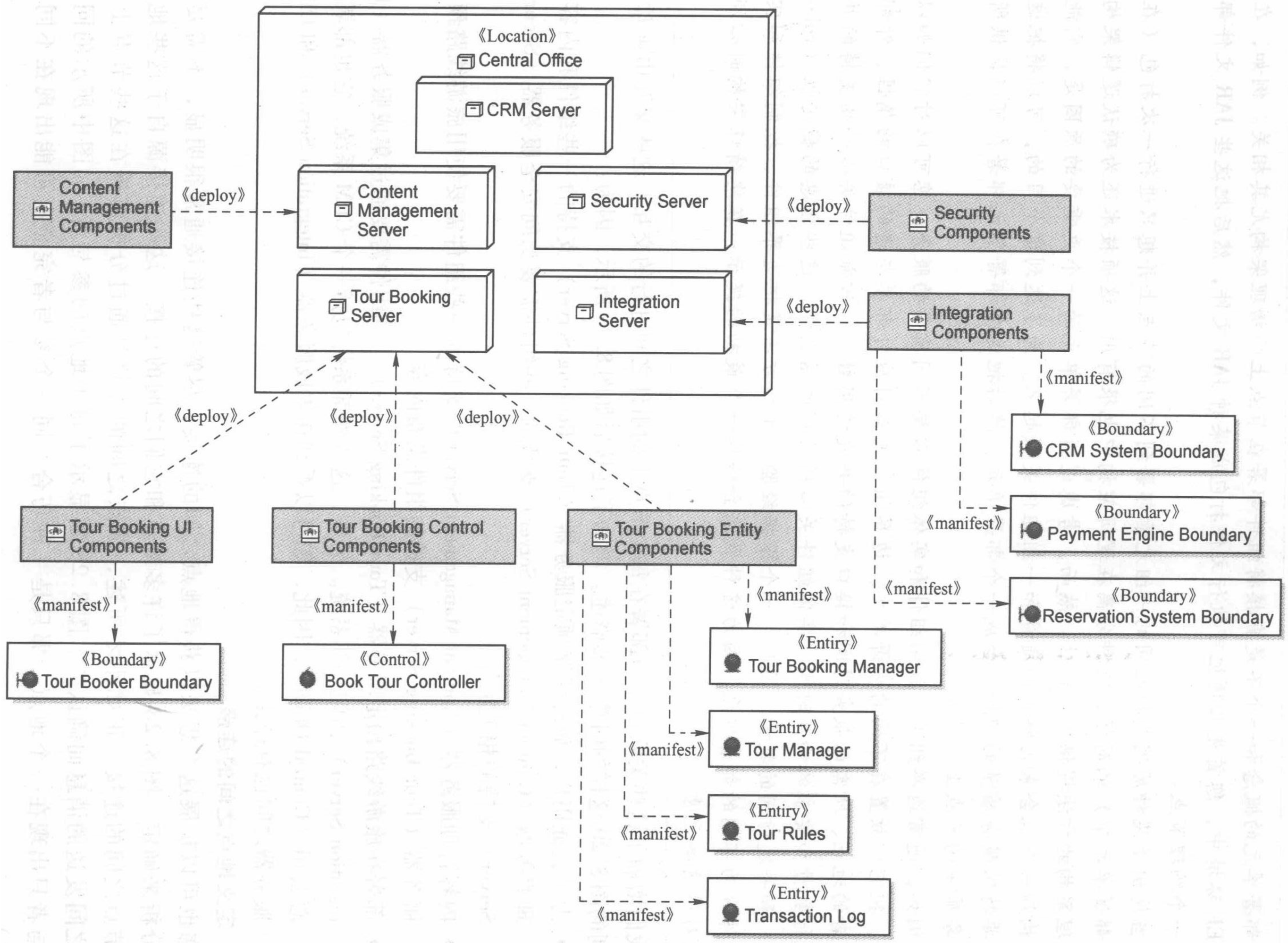


图 8.26　分配到总公司（Central Office）中节点的组件

部署单元的概念和一个不是直接将组件部署在节点上的物理架构尤其相关。例如，在Java EE环境中，您首先必须把可执行文件打包成某种JAR文件，然后把这些JAR文件部署到一个物理节点。

当您出于某种原因想把相同的一组元素部署到不同的节点上并想只进行一次打包（在一个部署单元中）的时候，这种分离在逻辑架构中也很有用。这项技术还为确认逻辑架构和物理架构的方法提供了一致性。然而，考虑逻辑部署单元的一个更重要的原因是，它能给架构师一个机会来把特定的质量和一组组件关联起来。为了达到这个目的，可以将把这些需要特殊服务质量的组件组合成一个部署单元，然后把这个部署单元部署在可以提供所需服务质量的节点上。

例如，通常边界组件、控制组件和实体组件需要不同质量的服务，您可以对它们加以区分，把它们放置在不同的部署单元。边界组件关心对应外部参与者的接口的属性、外部参与者的地点、所需的服务时间和接口支持的并发使用数量。控制组件关心必须支持的并发执行数量和要求的响应时间。实体组件关心所处理的信息量、这些信息的稳定性（volatility）和并发访问的数量。因此，一个根据经验产生的方法是把边界组件、控制组件和实体组件分为不同的部署单元，因为这种做法会给您一个最初的提示：在您精炼架构时必须区别对待这些元素。

我们增加了这些关联以作为部署在每个节点上的组件之间进行的交互。这些交互由本章前面提到的图8.20这样的时序图来描述。增加了连接后如图8.28所示，因为：

- 为了验证用户，旅游线路预定服务器（Tour Booking Server）支持的一些组件和内容管理服务器（Content Management Server）支持的一些组件需要访问安全服务器（Security Server）支持的相关组件。
- 内容管理服务器（Content Management Server）支持的一些组件需要使用旅游线路预定服务器（Tour Booking Server）支持的组件发布内容。
- 部署在旅游线路预定服务器（Tour Booking Server）上的组件需要通过集成服务器（Integration Server）访问外部系统。另外，这些外部系统包括一个CRM系统，它也部署在总公司（Central Office），因此，您可以看到在集成服务器（Integration Server）和CRM服务器之间也有连接。

步骤：定义地点之间的连接

您也可以扩展这个想法以获得地点之间的连接，尽管这些连接通常很明显，不需要仔细地分析来确定。图8.28显示了许多地点和它们之间的连接。这些连接源自于这些地点上的节点之间的连接。正如刚才所述，节点之间的连接是通过分析部署在这些节点上的组件之间发送的消息而确认的。图8.28还显示了每个地点的参与者。（图中所示的同一个参与者只出现在一个地点，那只是一种巧合；同一个参与者完全有可能出现在不同的地点。）

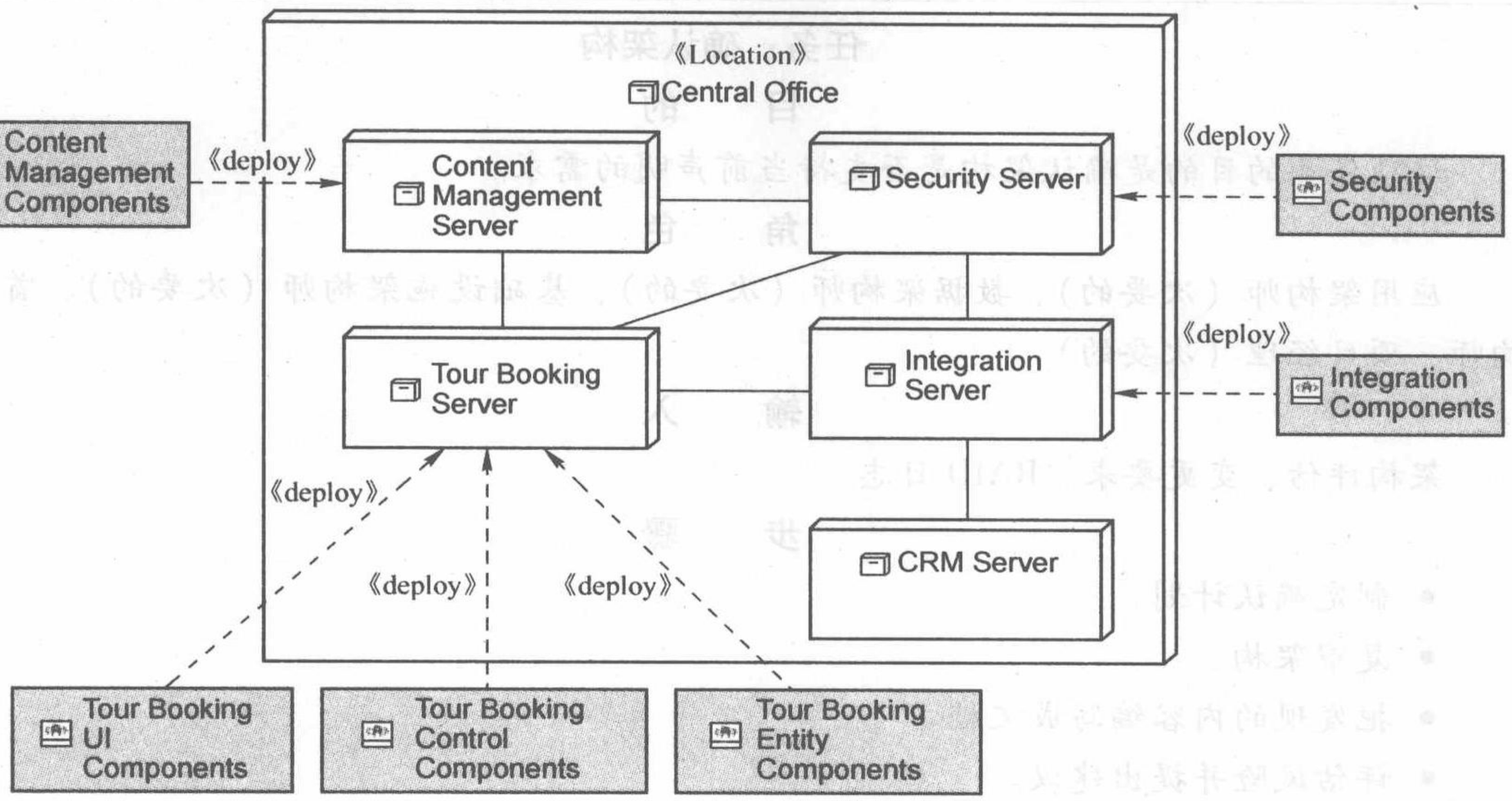


图 8.27　总公司内的节点和关联

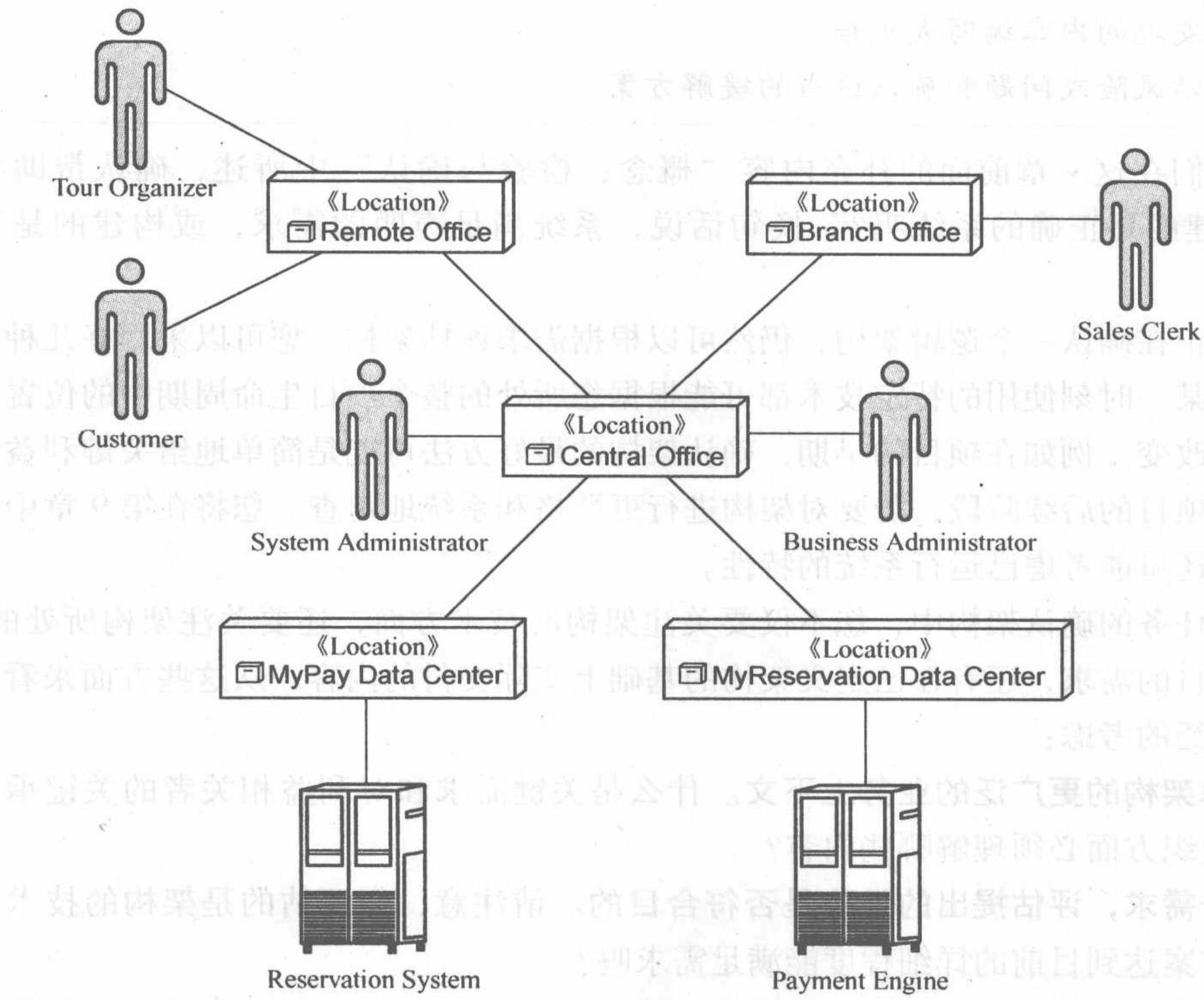


图 8.28　地点和连接

任务：确认架构

目　　的

这项任务的目的是确认架构是否支持当前声明的需求。

角　　色

应用架构师（次要的）、数据架构师（次要的）、基础设施架构师（次要的）、首席架构师、项目经理（次要的）

输　　入

架构评估、变更要求、RAID 日志

步　　骤

- 制定确认计划。
- 复审架构。
- 把发现的内容编写成文档。
- 评估风险并提出建议。

架构师角色

- 计划并领导确认。
- 把发现的内容编写成文档。
- 提示风险或问题和确认适当的缓解方案。

正如我们在这一章前面的补充内容“概念：检验与确认”中所述，确认帮助我们回答“我正在构建的是正确的系统吗?”换句话说，系统满足声明的需求，或构建的是其他的东西吗?

即使您正在确认一个逻辑架构，仍然可以根据需求评估架构。您可以采用好几种方法确认架构，您在某一时刻使用的特定技术都可能根据您所处的整个项目生命周期中的位置和当前架构的状态而改变。例如在项目的早期，确认架构的最好方法可能是简单地给关键利益相关者描述架构。在项目的后续阶段，需要对架构进行更严格和系统地走查。您将在第 9 章中看到物理架构的确认还可能考虑已运行系统的特性。

在这项任务的确认架构中，您不仅要关注架构的技术方面，还要关注架构所处的上下文，包括架构源自的需求，还有在已定义架构的基础上实际交付的内容。从这些方面来看，这项任务包括很广泛的考虑：

- **理解架构的更广泛的业务上下文。**什么是关键需求和对利益相关者的关键承诺？在项目组织方面必须理解哪些内容?
- **基于需求，评估提出的架构是否符合目的。**请注意，您评估的是架构的技术方面。架构方案达到目前的详细程度能满足需求吗?
- **判断交付的能力。**架构除影响开发的内容和测试环境外，对开发系统需要的技术也有重大的影响，需要在完成项目需要的理想情况和现实情况之间权衡。这种特殊的考虑

突出了架构师和项目经理之间需要密切的联系。

- **确认和评估技术风险**。什么风险最突出，已经正式地管理这些风险了吗？如果这些风险变成了真正的问题，会对项目产生什么潜在的影响？如何减轻和控制这些风险？

在确认架构时，选取几个关键用例并走查它们以确保架构支持这些用例，有时候这很有用。后面的“概念：基于场景的确认”部分讨论了一个采用这种确认形式的方法。

确认可以采取很多种形式来进行，从架构师（或架构团队）对相关人员做介绍到进行一个跨越很多天的活动。在这项任务中概述的通用步骤试图概括这些形式。确认形式的正式程度取决于很多因素，其中当然包括解决方案的复杂程度、项目在整个项目生命周期中所处的位置和涉及的利益相关者的人数。

确认架构这个任务关注项目内部的利益相关者，如**项目经理**、**开发人员**和**测试人员**。另一方面，**和利益相关者复审架构**这个任务关注外部的利益相关者，他们不属于开发团队，如**应用所有者**和最终用户。

概念：基于场景的确认

为了采用分而治之的方法确认架构，下面这种做法通常会很有用：考虑一组关键的用于走查架构的场景及让架构师实际演示架构的场景，例如演示这几个方面：已经作出的决策、组成架构的元素和它们之间的协作、架构体现出来的质量。

一个众所周知的基于场景的方法是由 SEI（Clements 2002）提出的架构权衡分析方法（ATAM，Architecture Tradeoff Analysis Method），还有它的前身软件架构分析法（SAAM，Software Architecture Analysis Method）（Clements 2002）。ATAM 这样考虑一组场景：在每一场景中要么关注系统提供的功能，要么关注系统体现的质量。在任何一种情况下，这种方法为架构师提供机会在每一场景的上下文中表述架构。ATAM 包括**确认架构**（Validate Architecture）和**复审架构**（Review Architecture）这两项任务，我们在本章的其他部分阐述它们。

ATAM 的一个方面是架构评估，它遵循以下步骤：

1. **介绍 ATAM**。介绍这个方法，回答关于评估流程的任何问题。
2. **介绍业务驱动因素**。描述开发工作的商业动机和架构的关键驱动因素。这个步骤通常由业务人员、客户或项目经理进行。
3. **介绍架构**。架构师描述架构和它如何满足这些业务驱动因素。这个步骤通常通过考虑描述这种架构的各种观点来进行。
4. **确定架构方法**。在这个步骤中，采用的架构方法通常通过对架构师提问来确定。这些方法通常以高层架构决策的方式进行描述，例如架构风格和重要元素的选取。
5. **生成质量属性效用树**（quality attribute utility tree）。关键的系统质量被确定，根据场景来指定并排定优先级。这些场景精炼了任何可能比较模糊的质量，例如把性能放入具体的场景中以便可以测量，最终产生质量效用树。这个步骤的主要目的是确保与确认架构相关的场景被选取。“实际上，效用树告诉评估团队在哪里检查架构”（Clements 2002）。

6. **分析架构方法。**评估团队根据架构是否能够满足先前描述的每个排定优先级的场景中的业务驱动因素来确定架构所采用的方法是否适用。所有的风险都会标注出来。

7. **进行头脑风暴并定义场景的优先级。**与前面的步骤更多地考虑评估架构的技术场景相比，这个步骤考虑更大的场景并邀请所有的利益相关者来补充架构应该评估的场景。

8. **分析架构方法。**这个步骤等同于第6个步骤，但是不考虑任何新的场景。

9. **介绍结论。**对通过ATAM获得的信息进行总结并介绍给利益相关者。

步骤：制定确认计划

架构的确认应当制定合适的计划，因为这涉及确认关键的人员参与、安排需要的会议以及在会议前准备和分发资料（这项任务的输入）。确认是一项团队工作，首席架构师和项目经理应该都参与这个步骤。

步骤：复审架构

架构师向内部利益相关者介绍架构并回答任何问题。介绍通常从声明解决方案的业务驱动因素开始，然后架构师介绍架构。外部的利益相关者会在**和利益相关者复审架构**这个任务中进行考虑。

特别地，内部利益相关者应该想要看到架构如何满足系统需求的总的场景——这就是确认架构的主要目标。因此，架构师通常会强调高优先级的需求（功能性需求和非功能性需求）并指出介绍的架构如何满足这些需求。使用某种形式的追溯矩阵来显示架构元素如何能追溯到需求，这也十分重要。

复审架构的一个重要方面是确保利益相关者最初声明的所有需求在架构中都得到解决。就是在这儿，我们能够充分认识到可追溯性（从架构元素连接到需求）的价值。定义良好的可追溯性能够让复审者从架构追溯到需求来查看每一个需求如何得到满足。

在复审架构时，有一个检查列表来确保提出的问题都被解答且没有被遗漏，那会很有用。附录B提供了一些适当的检查列表，还提供了第4章中介绍的架构描述框架。这些检查列表还能够帮助架构师为这项任务做好准备，因为这些检查列表会提醒您在复审中需要回答的问题。

步骤：把发现的内容编写成文档

随着介绍的进行，可能发现架构中仍有很多项没有充分地说明，或者仍有问题没有回答。这些意见都应该记录下来以确保得到处理，发现的所有内容都应该记录在**架构评估**工作产品中。

如果很显然由于需求没有得到充分描述而导致不能针对声明的需求进行架构确认，发现的内容会如表8.7中所示。

表8.7 架构确认中发现内容的样例

发现	性能需求没有具体到可以测量。它们是不完整的、模糊的

步骤：评估风险并提出建议

把发现的内容编写成文档后，下一步就是判断与每个发现相关的有哪些风险。清晰地说明这些风险对项目的交付有哪些潜在的影响，这很重要。除了说明这些风险，使风险发生的可能性和风险真地发生后对项目的潜在影响量化，这也会有帮助。对于每一个风险，您还应该提出一些降低这种风险的陈述或建议，如表 8.8 中所示，可以做下面的任意一件事情：

- 通过修改解决方案来消除项目中的风险。
- 通过增加风险评估的可能性、缓解风险的任务、冗余性能或其他降低风险的措施来限制或减小风险。
- 接受风险而不作任何改变，但是当风险成真时，提供一个替代的计划或方法。

表 8.8 架构评估发现的内容、风险以及风险缓解方法的例子

发现	性能需求没有具体到可以测量。它们是不完整的、模糊的		
风险	从性能角度来说，系统无法按照各种利益相关者的期望进行交付		
可能性	高	影响	高
缓解方案	用精炼系统性能需求的观点来复审所有利益相关者的要求。必要的时候联系相关的利益相关者。根据精炼的性能需求来复审和精炼架构		

架构评估中确认的风险还应该记录在 **RAID 日志**中以确保在项目中可以追溯到它们。在评估期间，难点也应该确定并记录在 RAID 日志中。这项任务还可能会对已检查的工作产品产生一个或多个变更要求。

任务：更新软件架构文档

目　　的

这项任务的目的是把对架构影响重大的元素记录在软件架构文档中。

角　　色

首席架构师

输　　入

架构评估、架构决策、架构总览、架构概念证明、数据模型、功能模型

输　　出

软件架构文档

步　　骤

更新软件架构文档。

架构师角色

架构师负责这项任务。

步骤：更新软件架构文档

在这个步骤中，架构师更新**软件架构**文档中相关的意见。我们在第 4 章中讨论过**软件架构文档**的概览。根据那种结构，把来自已经创建工作产品的内容分别组装起来，如表 8.9 所示。

正如需求意见由需求相关的各种工作产品填充一样，您可以用各种架构相关的工作产品填充功能意见、部署意见和确认意见。这些工作产品就是本章中各项任务的输出。您还可以填充交叉意见，那也是架构描述框架的一部分。

表8.9 软件架构文档的各个部分和工作产品的对应关系

部　分	工作产品
架构的总览	架构总览
架构决策	架构决策
需求意见	请看第7章“定义需求”
功能意见	架构决策 架构总览 数据模型 功能模型
部署意见	架构决策 架构总览 部署模型
确认意见	架构评估 架构概念证明 复审记录 RAID日志
应用意见	这项意见具有交叉性，可能包括来自所有工作产品的元素
基础设施意见	这项意见具有交叉性，可能包括来自所有工作产品的元素
系统管理意见	这项意见具有交叉性，可能包括来自所有工作产品的元素
可靠性意见	这项意见具有交叉性，可能包括来自所有工作产品的元素
性能意见	这项意见具有交叉性，可能包括来自所有工作产品的元素
安全性意见	这项意见具有交叉性，可能包括来自所有工作产品的元素

当填充**软件架构文档**时，请记住，各种（详细的）架构工作产品中定义的架构和**软件架构文档**中定义的架构有所不同，这很重要。**软件架构**文档的目的是提供一个工具把所有与架构相关的信息放在一起以便交流和复审这种架构（这个文档有时候会被看作一个可交付物）。因此，这个文档包含的信息通常是所有架构工作产品中包含信息的一部分。

任务：和利益相关者复审架构

目　　的

这项任务的目的是为逻辑架构工作产品定义基线并使利益相关者认同当前详细程度的架构能够解决定义的需求。

角　　色

应用架构师（次要的）、数据架构师（次要的）、基础设施架构师（次要的）、首席架构师、利益相关者

输　入

软件架构文档、其他需要的工作产品

输　出

- 变更要求、RAID 日志、复审记录

步　骤

- 定义工作产品的基线。
- 聚集工作产品。
- 复审工作产品。

架构师角色

- 确保为正确的工作产品定义基线。
- 领导复审。
- 在复审过程中回答问题并提供澄清的注释。

和利益相关者复审架构的目的是确保他们关心的问题得到解决或会得到解决。逻辑架构或许没有解决所有的问题（也许一些问题要到定义了物理架构之后才能得到解决），但是它是解决所有问题的一个主要步骤。

> 架构复审是对架构的测试，是架构的一个安全网，按照基于风险的测试策略进行计划和执行。（Buschmann 2009）

我们建议在进行复审之前正式地为架构定义基线并对它进行变更控制，以便于从此以后可以按照这一基线控制所有影响架构的需求变更。关于在复审之前定义基线还是在复审之后定义基线存在一些争论。但是，我们的首选是先定义基线，在变更控制之下处理复审意见。这种做法可以防止由于复审发现了问题，为解决问题而形成新的架构，新的架构又需要复审这样的恶性循环而无法定义基线。已经存在很多关于架构复审的资料，但是，最权威的来源之一是《Software Architecture Review and Assessment（SARA）Report》（SARA 2002）。

步骤：定义工作产品的基线

定义工作产品的基线非常简单，包括选取代表当前架构（也是当前活动的输出之一）的工作产品集，确保把它们加入配置管理系统并赋予正确的版本号。然后，您需要发布这组作为基线的工作产品集以便它们可以供后续的活动引用，例如进行详细设计或定义物理架构（我们会在第 9 章中讨论它）。

步骤：聚集工作产品

架构复审活动的主要输入就是**软件架构文档**。然而，其他工作产品都可以用作复审期间的参考，包括产生架构的需求工作产品。正如我们在先前的任务中所述，**软件架构文档**的内容来自于各种架构相关的工作产品，也可以复审这些工作产品。

步骤：复审工作产品

在这个步骤中，您将复审这些架构工作产品。复审可能仅仅是走查**软件架构文档**的内容，

如果需要，还参考可用的支持文档。发现的任何问题都记录在**变更要求**中。当复审完成后，您要在**复审记录**中概要地记录复审结果，包括任何行动项。任何必须上升为项目问题的东西应该记录在项目 **RAID 日志**中。

复审本身可能会持续几小时或几天（或更长），这取决于复审的架构。复审通常是检查架构的很多方面，类似于进行确认，包括架构目标的澄清（与声明的需求一致）、对优先级的认可、检查所作的折中（如灵活性与性能）和对风险的考虑。

复审作为一次迭代的质量关，确保逻辑架构足够稳定以支持接下来的任务，如逻辑架构详细设计、制定测试计划概览或充实项目计划。因此，相关各方的参与很重要，例如包括**项目经理、测试人员**和**开发人员**。另外，应用程序的拥有者会有兴趣得到架构的总览并理解架构如何满足关键的战略性需求。用户需要保证关键的业务需求获得满足。

8.5 总结

定义逻辑架构是整个软件开发过程的关键部分。如果进行得好，会产生比较健壮和易于理解的架构，能够清晰地分离各种问题并平衡系统元素之间职责的划分。定义逻辑架构也是帮助您从系统需求通往物理架构的跳板。这一章讨论的**创建逻辑架构**活动将引起**创建逻辑详细设计**活动（我们在第 1 章“导言”中有所讨论），在这个活动中，我们会补充在逻辑层面需要的任何余留细节，例如一个特定业务规则的细节。

在下一章，我们将阐述如何使用逻辑架构（和逻辑详细设计）来得到一个直接支持后续开发活动的物理架构。

第 9 章

创建物理架构

在这一章中，我们从第 8 章中讨论的逻辑架构转向创建相应的物理架构。不同于它的逻辑架构，物理架构接受将用于详细物理设计及最终编码的所有产品和技术。

正如在第 4 章中所述，一个模型在它的生命周期中按照离散的实现层级发展，从内容非常概念的最初模型一直到内容详细到可以作为实现基础的模型。在这一章中，我们介绍两个关键的用于描述系统架构的模型（**功能模型**和**部署模型**）如何从只包含纯逻辑（或独立于技术）的元素演变到包含物理（或特定技术）的元素。

在这一章中，我们会描述用于创建物理架构的各项任务，这种物理架构用于实现第 7 章中定义的需求并发展第 8 章中定义的逻辑架构。我们假设有几种工作产品流入或流出**创建物理架构**活动，如图 9.1 所示。正如您从这张图中看见的，物理架构的考虑既涉及需求工作产品又涉及逻辑架构工作产品。

9.1 从逻辑架构到物理架构

由于逻辑架构是您的主要输入之一，因此当从逻辑架构转向物理架构时，应该理解这个工作产品意味着什么。除了考虑系统的需求，您还必须考虑逻辑架构中定义的各种元素，如图 9.2 所示。这张图既包含功能元素也包含部署元素，还显示了从逻辑架构通往物理架构的过程中需要考虑的各项因素，包括组件、位置、节点、节点连接和部署单元。为了简化，这个图中没有显示子系统、组件接口、操作和各种关系。我们会在这一章后面讨论这些各种元素。

在从逻辑架构转向物理架构的过程中，您会作出好几项选择：

- **技术**。这项选择包括用于定义运行环境的平台，例如 Java 平台企业版（Java EE）或微软的 .NET，还有用于实现任何组件的编程语言。
- **产品**。这项选择包括支持任何组件实现及运行的产品，例如应用服务器和数据库服务器，还包括硬件的选取。如果需要了解产品在当前上下文中的含义，请参考补充内容“概念：产品”。
- **可重用资源**。软件资源（正如第 5 章“可重用架构资源”中所述）包括封装的应用程序、现有的组件和所有的遗留系统。您还可以选择当前 IT 环境中可用的硬件。
- **定制开发**。由于各种原因，您可能决定定制构建某些组件。定制构建（Custom-built）的组件提供了最大的灵活性，因为它们完全由您控制。这类组件通常是为了实现一组特定的功能，而且，如果您只需要产品所提供的一小部分功能，那么，定制组件会比购买产品更便宜。在某些情况下，也许还会构建定制的硬件。

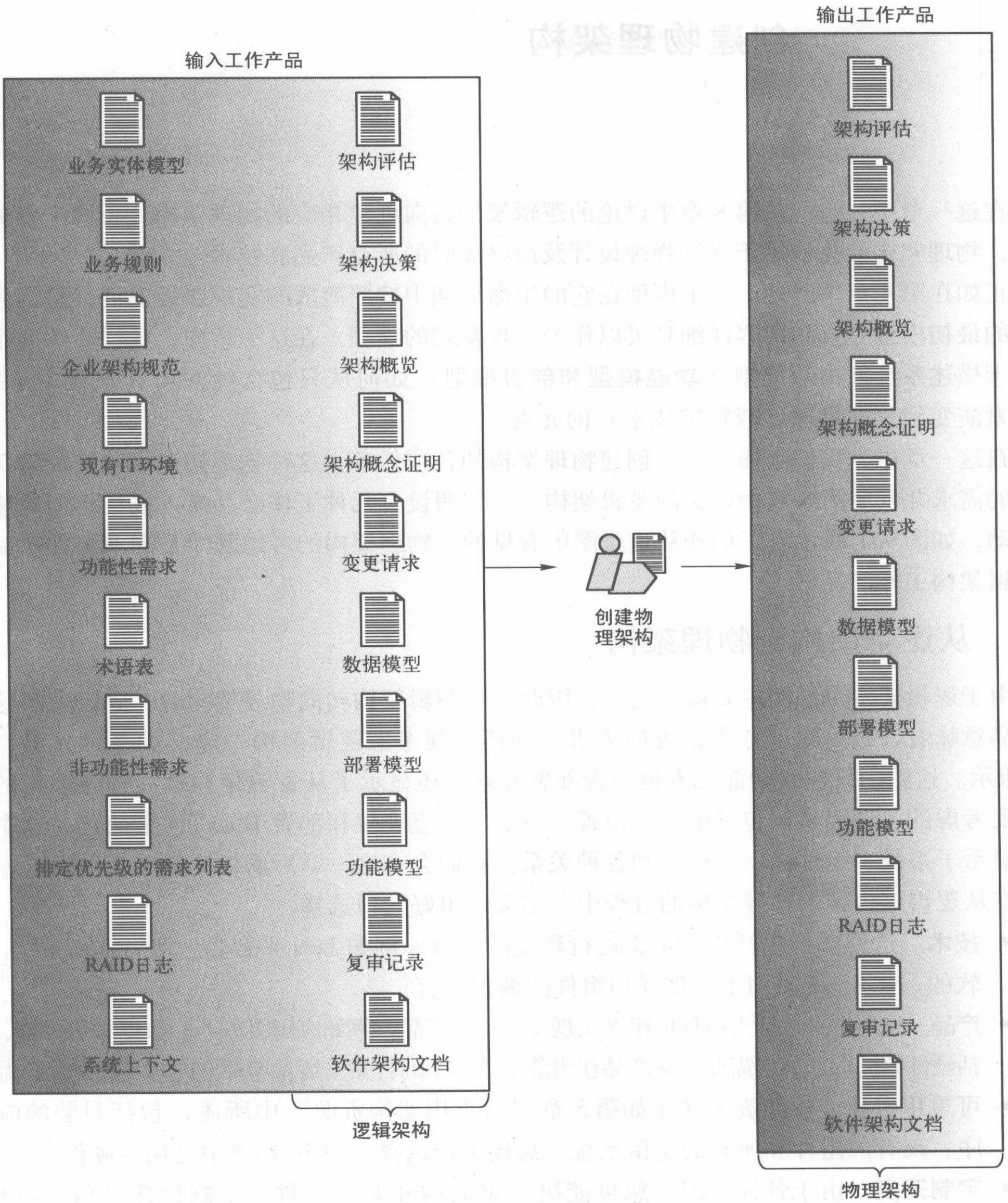


图 9.1 创建物理架构活动的输入和输出工作产品

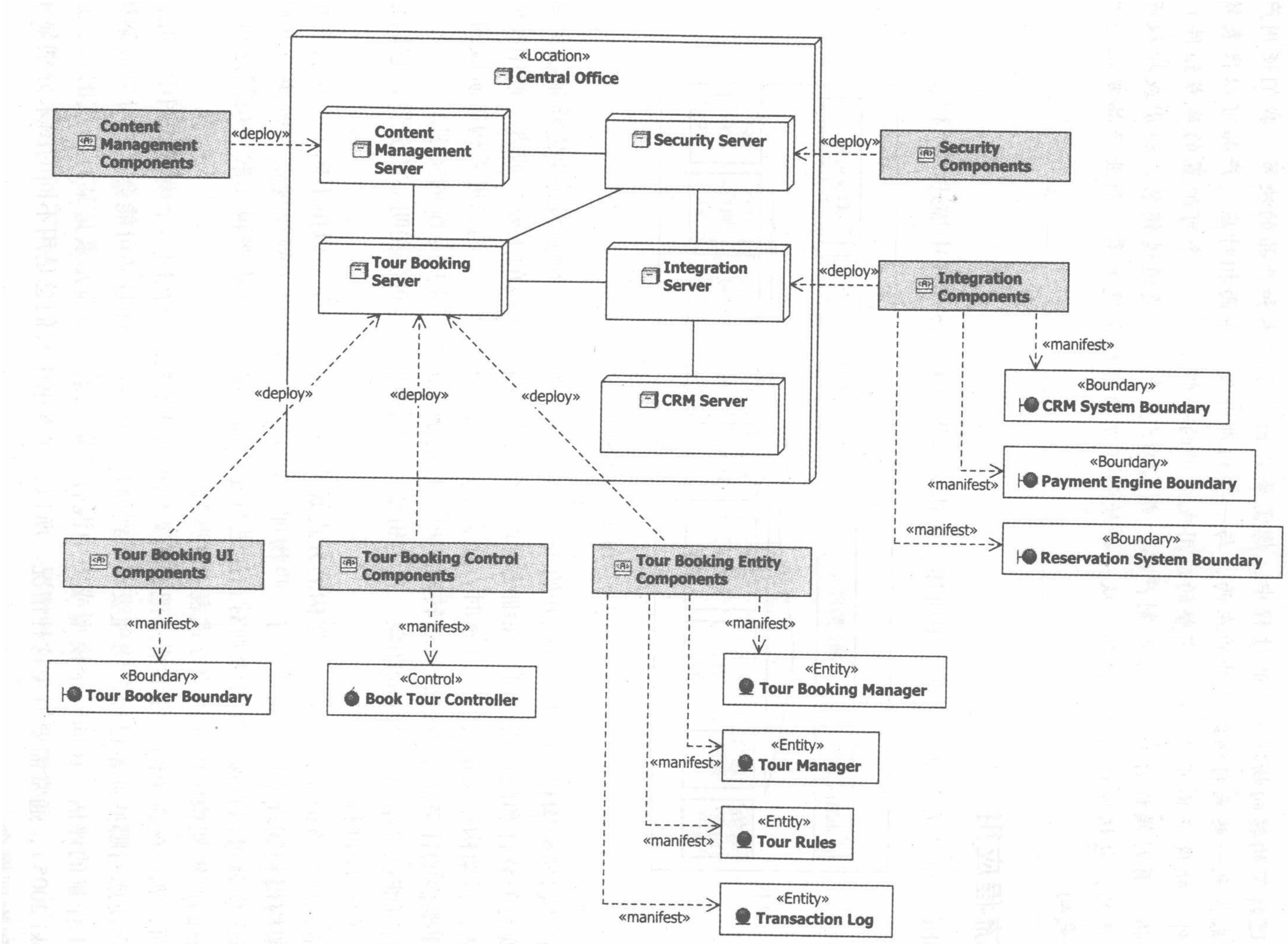

«Location»
Central Office
Content Management Components
«deploy»
Content Management Server
Security Server
«deploy»
Security Components
Tour Booking Server
Integration Server
«deploy»
Integration Components
«deploy»
«deploy»
«deploy»
CRM Server
«manifest»
«Boundary»
CRM System Boundary
«manifest»
«Boundary»
Payment Engine Boundary
«manifest»
«Boundary»
Reservation System Boundary
Tour Booking UI Components
Tour Booking Control Components
Tour Booking Entity Components
«manifest»
«manifest»
«manifest»
«Boundary»
Tour Booker Boundary
«Control»
Book Tour Controller
«Entity»
Tour Booking Manager
«manifest»
«Entity»
Tour Manager
«manifest»
«Entity»
Tour Rules
«manifest»
«Entity»
Transaction Log

图9.2　部分逻辑架构

概念：产品

从逻辑架构转向物理架构的过程中，您需要作出关于技术和产品的决策。我们使用**产品**一词来代表一些来自组织之外的东西，与一些内部开发的东西相对应。产品可以代表软件或硬件。软件产品可以是一个完整的应用或应用的一部分（如一个可配置的库或组件）。商业产品（有时候称为商业的非定制产品，或 COTS、产品）是通过销售、租用或授权而得到的产品，包括授权的开源产品。从一个软件开发项目的角度来看，商业产品要求您考虑产品采购。

9.2 流程应用

正如第3章“方法基本原理”中所述，架构规范大致遵循图9.3中描述的流程。

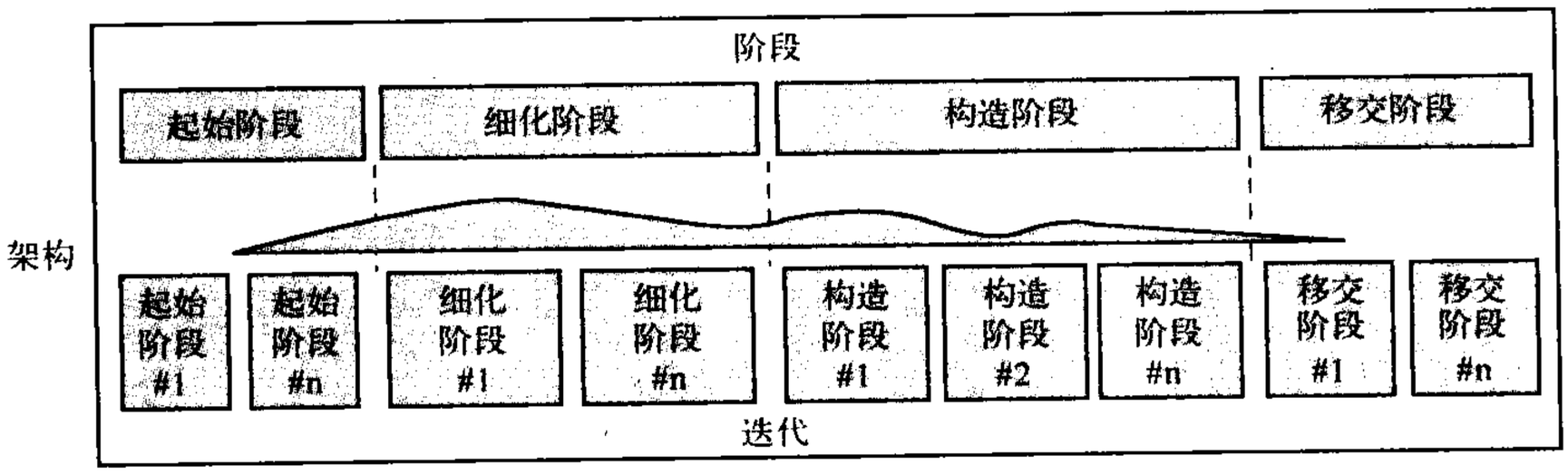


图9.3 物理架构和迭代开发

与定义物理架构相关的任务可以开始于起始阶段的一个迭代，例如当您决定将解决方案基于一组现有的软件和硬件资源时。在细化阶段，当您专心架构系统的主要（或者，使用我们常用的说法，对架构意义重大的）元素时，您会把这些任务作为重点。当您进入构造阶段的迭代时，物理架构的任务逐渐减少，焦点转向了根据目前定义的架构完成系统的实现。

我们可能已经给出了这样的印象：在逻辑架构和物理架构之间存在明显的分界，它们总是按照单一的顺序进行。

但是，您应该记住，形成一个架构的方式既可以由上向下，也可以由下向上。尽管您可以从逻辑架构得到物理架构，但是，有一些物理元素可能是指定的（例如规定为解决方案的约束），那么您就可以直接在物理级别对它们进行建模而不必首先考虑逻辑元素（虽然您可以选择在任意的逻辑架构中以抽象的方式表示这些物理元素）。

然而，您应该避免的一件事情是把逻辑架构和物理架构混合在同一个模型或图中。混合这些概念会使您对哪些元素已经在物理级别得到处理感到疑惑，而且还可能会扭曲对完成架构产品所需工作量的评估。在同一个模型或图中混合逻辑元素和物理元素是某种形式的观点过载（Rozanski 2005），通常需要避免这种情况。所以，在本书中我们总是用不同的图来分别显示逻辑概念和物理概念。

9.3 创建物理架构：活动概览

创建物理架构仅仅是架构任务的一个特定关注点，因此，这些任务本身和用于定义逻辑架构的任务非常相像并不奇怪，如图 9.4 所示。仅有的不同是这些任务关注的是开发系统的物理方面而不是逻辑方面。

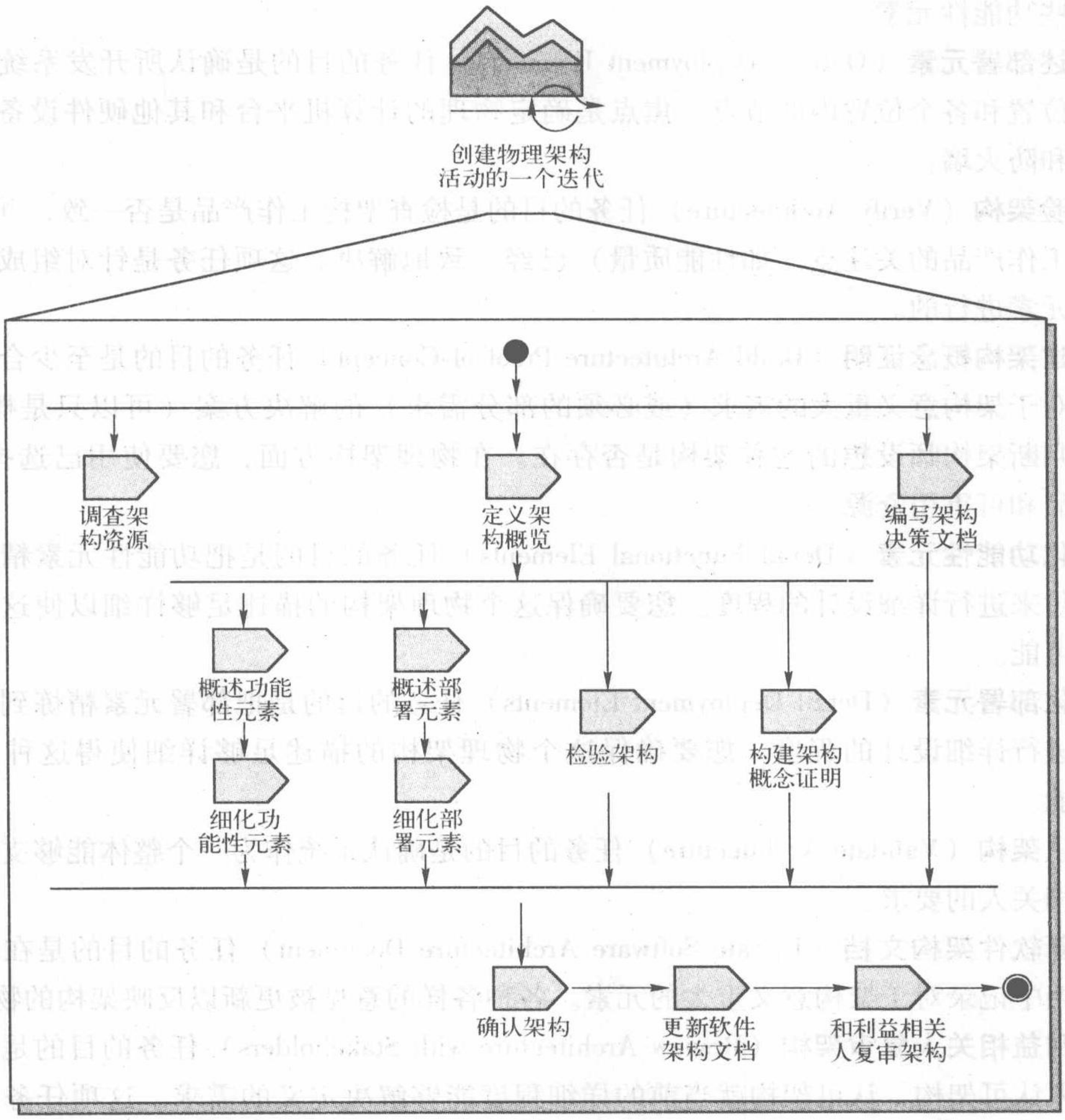


图 9.4 创建物理架构活动的概览

下面是对这些任务的总结，主要关注物理架构的创建：

- **调查架构资源**（Survey Architecture Assets）任务的目的是确认可用于所开发系统的可重用架构资源。您要以已经定义的需求和逻辑架构为基础进行调查。
- **定义架构概览**（Define Architecture Overview）任务的目的是确认并描述所开发系统的主要元素。物理架构反映了对关键技术、产品、可重用资源的选择。

- **编写架构决策文档**（Document Architecture Decisions）任务的目的是记录架构形成过程中所做的关键决策和决策背后的原因。创建物理结构中的重点是与选择关键技术、产品、可重用资源相关的决策。
- **概述功能性元素**（Outline Functional Elements）任务的目的是确认所开发系统的主要功能性元素（子系统和组件）。焦点是如何通过产品、可重用资源和定制的软件元素实现这些功能性元素。
- **概述部署元素**（Outline Deployment Elements）任务的目的是确认所开发系统将要部署的位置和各个位置内的节点。焦点是确定物理的计算机平台和其他硬件设备，如路由器和防火墙。
- **检验架构**（Verify Architecture）任务的目的是检查架构工作产品是否一致，并确保跨架构工作产品的关注点（如性能质量）已经一致地解决。这项任务是针对组成物理架构的元素进行的。
- **构建架构概念证明**（Build Architecture Proof-of-Concept）任务的目的是至少合成一个满足对于架构意义重大的需求（或必须的部分需求）的解决方案（可以只是概念性的）以判断架构师设想的这种架构是否存在。在物理架构方面，您要使用已选择的技术、产品和可重用资源。
- **细化功能性元素**（Detail Functional Elements）任务的目的是把功能性元素精炼到可以交付来进行详细设计的程度。您要确保这个物理架构的描述足够详细以使这种交付成为可能。
- **细化部署元素**（Detail Deployment Elements）任务的目的是把部署元素精炼到可以交付来进行详细设计的程度。您要确保这个物理架构的描述足够详细使得这种交付成为可能。
- **确认架构**（Validate Architecture）任务的目的是确认系统作为一个整体能够支持各种利益相关人的要求。
- **更新软件架构文档**（Update Software Architecture Document）任务的目的是在软件架构文档中记录对于架构意义重大的元素。各种各样的意见被更新以反映架构的物理方面。
- **和利益相关人复审架构**（Review Architecture with Stakeholders）任务的目的是使利益相关人认可架构，认可架构就当前的详细程度能够解决定义的需求。这项任务明确地意味着获取对物理架构充分性的认可。

尽管在本章中我们关注的是物理元素而不是逻辑元素，但是，本章本质上重复了我们在第8章“创建逻辑架构”中定义的任务，所以，我们在本章中没有包括全部任务的描述。因此，读者应该假设所有的角色、工作产品和步骤与上一章相同，除非我们另外说明。当我们在本章中讨论每个任务时，我们会引用任何已经创建且您也可能已经掌握的逻辑元素。

在第8章中，您已经看到关注功能性元素和部署元素的任务分为概述任务和细化任务。我

们说过，概述任务关注对架构来说意义非常重大且由项目的首席架构师负责的元素。细化任务关注略微次要的元素，由拥有相应关注点的架构师负责——例如系统架构师、基础设施架构师或数据架构师。表 9.1 概述了这一章中考虑的物理元素。

表 9.1　概述任务和细化任务中考虑的元素

元　素	概　述	细　化
功能性	子系统	界面
	组件	映射到数据
	操作	操作签名
部署	位置	部署单元
	节点	连接

9.4　任务：调查架构资源

当创建逻辑架构时，您会关注有助于定义基本结构和架构元素之间交互的资源。当创建物理架构时，您会寻找有助于实现逻辑结构和逻辑元素之间交互的物理资源。从这个角度看，逻辑架构充当了一组需求和您可能选取来实现架构的物理资源的约束。

正如在第 8 章“创建逻辑架构”中所述，这项任务通常会在一个迭代过程中被执行好几次，因为在创建物理架构的过程中，架构资源的重用始终是架构师头脑中首先想到的。这种重用经证明有很多类型的可重用资源可以考虑，涉及从一个特定技术的参考架构到一个特定技术的设计模式。

举一个简单的例子，假设您已经决定 YourTour 系统将拥有一个独立的内容管理子系统，这个需求会分析并记录为相关功能元素的职责。当前，您正把一个单独的内容管理组件分配给这个内容管理子系统。如果您决定必须进一步分解这个子系统以把它外包给不同的供应商（例如数据库和内容管理工作流分离），那么，这样的分配就发生变化。需要一个内容管理解决方案在逻辑**架构概览**和逻辑**功能模型**中都已经确认了。此外，您已经确定了这个组件的非功能需求。表 9.2 概述了这个组件的特征。

表 9.2　内容管理组件的组件描述

组 件 名 称	内容管理组件
描述	用于创建、编辑、搜索和发布各种数字媒体，包括文本文件、视频文件、音频文件和任何其他形式的 Web 内容；也用于保存内容和进行版本管理。这个组件还支持识别 YourTour 系统用户的内容管理角色的能力，也支持为不同类型的内容管理分配角色的能力
职责	支持不同的用户角色创建、更新和删除内容 支持把不同的角色分配给不同内容类型的能力 支持使用工作流创建内容 在创建内容的不同角色之间支持使用工作流 对所有的内容类型支持多版本 支持内容语义和显示方式的分离 支持从外部抓取内容（包括扫描和 Web） 支持不同来源内容的发布（包括 HTML、XML、PDF）

（续）

组件名称	内容管理组件
与这个组件相关的非功能性需求	根据 YourTour 组织的标准，系统将支持存在视力障碍、听力障碍、行动障碍或感知障碍的人使用 对于需要安全访问的功能（例如预定旅游线路），系统的可用率将达到 99.9%。对于其他的功能（如浏览旅游线路），系统的可用率将达到 99%。备份和维护操作将不要求系统停机。 系统将使用业界标准的 Web 协议实现交互操作 YourTour 系统将可以通过上网设备进行访问，例如通过 Web 浏览器和 PDA

基于这个内容管理组件的描述，您会寻找一个或多个尽可能多地满足这些职责和质量的现有资源，并拟定一个潜在的备选资源列表。很明显，如果没有找到合适的资源，您可以去看看可作为这个组件的现有商业软件产品。如果没有找到合适的备选资源，您应该考虑创建自己的组件。

调查架构资源任务既考虑软件也考虑硬件，记住这一点很重要。因此，您同样需要考虑从**现有 IT 环境**工作产品（包括现有的系统和基础设施）中确定的可重用资源。

9.5 任务：定义架构概览

这项任务的目的是确认并描述开发系统的主要元素，这些元素反映了所有已选的技术、产品和可重用资源。

当您用特定的技术元素实现**功能模型**（Functional Model）和**部署模型**（Deployment Model）时，在物理架构的**架构概览**中反映这些元素可能会有用。相对于逻辑架构概览，这种架构概览提供了最初设想的物理架构的草图。

图 9.5 显示了描述 YourTour 系统物理元素的架构概览，其中（正如您将看到）系统已经决定基于 Java EE 技术。（图中没有明确地显示使用 Java EE，只显示了部署元素）。正如您可以从这张图中看到的，更新的架构概览受到了逻辑架构的影响，这时显示的是物理节点，而不是逻辑子系统。

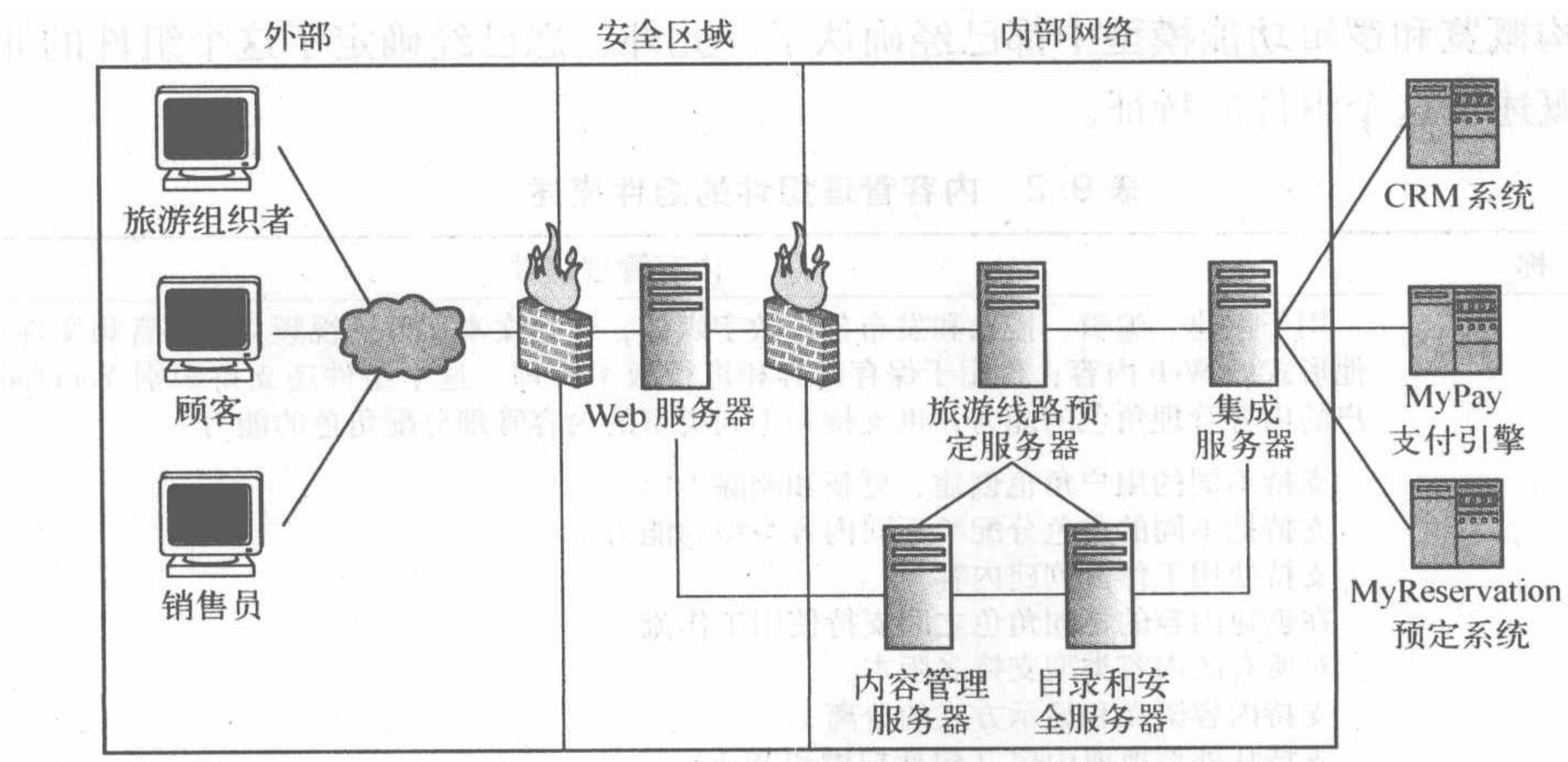


图 9.5 YourTour 系统物理架构概览

图 9.5 中显示的连接需要特定的安全性考虑。图中也包括了防火墙来表明 YourTour 系统设定的安全区域。这些防火墙的放置遵循了那些可以通过 Web 访问的系统的标准模式（IMB 2009），第一个防火墙保护整个系统，另一个防火墙为内部网络内的元素提供额外的安全保护。

当创建描绘物理架构的**架构概览**时，您应该参考驱动逻辑架构的需求，还要参考描绘逻辑架构的**架构概览**。尽管逻辑架构提供了第一个近似于要求的架构，但是，也许并没有考虑所有的需求——特别是强制使用特定技术或产品的约束。另一方面，这个描绘物理架构的**架构概览**确实承认这些约束，这就是为什么您在图 9.5 中看到了对 MyPay 支付引擎和 MyReservation 预定系统的引用；这些元素就是对解决方案指定的约束。补充内容“概念：处理非功能性需求”中专门讨论了处理物理架构中的非功能性需求。

概念：处理非功能性需求

影响物理架构的主要因素之一是为处理非功能性需求（包括质量和约束）所采用的方法。尽管在逻辑架构中架构师采用了特定的步骤处理这些需求，但是，这些需求在物理架构中会得到更清晰和全面的处理。

特别地，许多方法都采用了比较理想化的方式获得逻辑架构，其中网络的速度无限快；计算机从不崩溃；处理器的能力、磁盘空间和内存都无限大。然而，在定义物理架构时，您无法避开这些问题。当您在物理架构的运行环境中考虑它时，您已经描绘的任何“最佳”架构都会很快被带入现实。

架构概览仅仅是为和各种利益相关者交流而提出的对架构的一个总的看法。它并不是一个可以作为系统构建基础的严格的架构规范。而是更具体的**功能模型**和**部署模型**的目的，我们将在这一章接下来的任务中关注它们。

9.6 任务：编写架构决策文档

当考虑物理架构时，架构师会针对特定的技术、产品和可重用资源等的选取做出决定。特别地，架构师会做出购买还是构建的决定（我们会在这一章后面的**概述功能性元素**任务中更详细地讨论）。充分地记录这些决策来反映所有考虑过的选项和选择背后的原因，这十分重要。

当进行这些选择的时候，您有好几项有用的输入。**RAID 日志**帮助您关注需要做出决策的范围。这些架构模型（就是**数据模型**、**部署模型**和**功能模型**工作产品）确保任何决策都在架构演化的上下文中进行。**架构评估**和**架构概念证明**工作产品是帮助选择正确的选项并最终做出决策的很多信息的来源。

9.7 任务：概述功能性元素

这项任务关注于通过确定子系统和组件的物理部分来实现逻辑子系统和逻辑组件。

9.7.1 将逻辑功能元素映射到物理功能元素

这项任务关注于通过确保适当地实现已经确认的逻辑子系统和逻辑组件来确认将构成物理架构的子系统和组件。

在第8章“创建逻辑架构”中，子系统定义为一组相关的组件。这一个定义对于物理架构同样有效，物理子系统代表一组相关的物理组件。逻辑子系统到物理子系统的映射通常相当简单直接，一般是一对一。但是，我们可能会额外构建一些反映特定技术考虑的子系统。例如，我们可能确认了一组没有和其逻辑部分一一对应的物理组件，创建额外的子系统放置这些组件是有意义的。

物理子系统也可以代表一个把好几个组件绑定在一起的产品，尽管是由这些组件松散耦合而成。例如，如果您已经选择购买一个客户关系管理系统（CRM）子系统而不是使用现有的CRM系统，您可能会发现这个CRM系统包含几个独立的组件，分别支持联系人管理、财务、订购管理等（您已经单独考虑的元素），它们作为一个完整的子系统一起出售。

就组件而言，逻辑组件和实现的物理组件并不总是一一对应的，下列的对应方式都有可能：

- **一对一**。如果分配给一个逻辑组件的职责完全可以由一个物理组件来完成，那就可以用单独的一个物理组件来实现这个逻辑组件。例如，YourTour系统的内容管理组件完全可以映射到一个产品。
- **一对多**。例如，如果当您不能只通过一个产品或打包应用来实现一个逻辑组件而不得不购买和构建部分组件的时候，那么，您可以使用多个物理组件来实现这个逻辑组件。在这种情况下，您必须把职责分配给多个物理组件并提供一个在这些组件之间通信的方法。
- **多对一**。您可以使用单个物理组件来实现许多逻辑组件，例如，当您只有把两个逻辑组件合并为一个物理组件才能满足性能需求的时候。

上述例子是从逻辑组件变为物理组件时它们之间的对应、分割和聚合关系的特例。补充内容“检查列表：把逻辑组件映射到物理组件”中提供了一个更完整的检查列表，它包括所有需要考虑的因素。

检查列表：把逻辑组件映射到物理组件

当出现下列情况时，我们考虑在逻辑组件和物理组件之间一对一的映射：

- 单个物理组件可以支持对应逻辑组件的全部职责，利用适当的实现技术而不破坏为组件分配职责时使用的耦合、内聚和粒度。
- 单个组件可以支持任何非功能性需求而不需要其他一对多或多对一的映射。

当出现下列情况时，我们考虑在逻辑组件和物理组件之间一对多的映射：

- 需要多个物理组件来实现这个逻辑组件。这些物理组件可能由好几个产品和定制构建的组件组成。

- 需要多个物理组件来支持已经分配给这个逻辑组件的不同服务级别的特性。例如，在 YourTour 案例中，预定一个旅游线路可能需要 24 个小时，而取消一个旅游线路通常只需要一个正常工作日，因为这个任务需要预售员额外的确认和检查。
- 需要使用多个物理组件来实现一个已选择的、要求把分配给一个逻辑组件的职责在多个物理组件中实现的架构模式。

当出现下列情况时，我们考虑在逻辑组件和物理组件之间多对一的映射：

- 单个物理组件（如一个产品）封装了分配给多个逻辑组件的职责。
- 单个物理组件支持多个逻辑组件需要的一个服务级别的特性。散布于多个逻辑组件的业务规则需要放置在同一个地方，例如，一起放在单个规则引擎组件中。
- 实现多个逻辑组件的单个物理组件需要适应已选择的特定架构模式。

逻辑架构尽可能地确保组件和技术无关，因此，能够使用不同的技术和产品来实现逻辑架构。在实际情况中，当选择了实现逻辑架构的技术和产品之后，必须记住每项技术都有自己的特性，这意味着您必须对这些差异进行一些解释以确保组件的说明没有歧义，从而使得这些组件可以进一步细化和实现。然后，物理架构就成为设计人员和开发人员遵循的一个蓝图。

一个简单的例子是考虑用某种编程语言实现的接口。您可能已经花了很多精力在逻辑层面说明接口以帮助自己在选择物理元素时做出比较充分的决策。您还定义了在逻辑元素和物理元素之间保持一对一的对应关系，还希望尽量地在物理架构中保持这些接口。虽然有些语言明确地支持接口（如 Java 和 C#），但有些却不支持（如 C ++、C 和 COBOL），因此，您需要制定一个方案以便在这些语言中支持接口。

另一个需要考虑的问题是所选技术如何处理异常、事务、持久、集成等。这些方面通常描绘架构的机制，我们在第 5 章“可重用架构资源”讨论的资源类型中包含了这些机制。

9.7.2 确认物理功能元素

下面是一个可以用来确认物理子系统和物理组件的显而易见的顺序：

1. 调查包含在**功能模型**中的所有逻辑子系统和逻辑组件，决定每一项是否使用现有的资源和产品来实现，是否需要构建。

2. 对于采用资源和产品实现的每个组件，联系**项目经理**启动采购流程来购买那个产品。

3. 对于将要构建的每一子系统或组件，决定实现的技术并提供足够的指导以确保组件的详细设计和实现能够顺利进行。

在这里，我们使用术语**购买**来表示采购任何已经存在的东西，包括第 5 章“可重用架构资源”中讨论的所有类型的资源和来自商业软件或硬件供应商的产品。组织中可能已经存在某些资源（也就是说，它是另一个项目的开发成果或公司的可重用程序）。关于决定购买还是构建的指导，不同组织和不同项目之间的策略变化很大。在决定购买还是构建的过程中，复审一下补充内容“检查列表：购买与构建决策的比较”中的内容是值得的。

我们强烈建议架构师参与采购决策，特别在需求复杂并面临几种选择的情况下，维护正在开发的系统的技术完整性很重要。架构师知道系统未来的全貌，因此，架构师是参与这个过程并确保这种完整性的最佳人选。

当存在几种选项时，在**架构决策**工作产品中记录这些选项和最终决定的理由是值得的。请记住，当您考虑这些选项时，同时查看功能性标准和非功能性标准很重要。有可能一个产品或资源看起来可以满足您所有的功能性需求，但性能和可靠性等质量可能非常糟糕，特别是如果这种产品或资源还不成熟或在投放市场不久的时候。

检查列表：购买与构建决策的比较

当决定**购买还是构建**的时候，可以考虑下列问题。在这个检查列表中，术语**产品**通常指组织外的事物，但是，在此表中，我们也用这个术语来表示组织内的可重用资源，因为当复用其他团队或业务单位生成的资源时，您需要问许多相同的问题。

- 公司普遍的政策是倾向于购买还是构建（或其他）？
- 如果不存在可以完全满足需要功能的产品时，更改需求以便可以重用一个产品是否值得考虑？
- 如果一个现有的产品在可以使用之前需要很多的客户化工作，改为采用定制的解决方案是否更好？
- 需要的组件是最先进或独一无二的吗？如果是，可能不稳定或不成熟吗？
- 产品供应商的财务可靠吗？
- 产品的供应商提供哪些支持选择？
- 如何向供应商提出问题并把问题不断升级？应该提供哪种级别的服务协议（SLA）（如问题修复的时间）？
- 使用现有产品（包括升级）产生的维护费用存在财务问题吗？
- 有没有考虑维护成本，有没有计算拥有产品的总成本？
- 产品供应商的移植策略（例如迁移数据时尽量不要中断正在进行的操作）可接受吗？
- 产品的集成需要难于掌握的专业技能吗？
- 产品的客户化工作非常困难和耗时吗？会导致进度延期吗？
- 产品必须依靠其他产品或平台吗？已经把这些产品或平台放入决策流程中了吗？
- 具有正确版本号的产品能和其他可能使用的产品兼容吗？（供应商经常发布一些至少在短期内和其他产品（可能是它们自己的产品或其他供应商的产品）不兼容的新版本产品）。
- 产品有值得信赖的（和成功的）使用案例吗？
- 供应商有该产品的长期发展方针吗？
- 供应商是否值得信赖并符合您作为用户的目标？

- 产品的发布时间表可以接受吗？
- 是否有任何免费的产品可用，例如未授权的开源软件？（尽管对于企业来说，使用这类软件构建关键任务系统风险过高，但是，这类软件可能是由专业的在线开发团队开发的，可能具有比商业软件更高的质量。对于特定类型的系统，这个办法非常有利。当使用开源软件时，您应该知道，即使许可证可能是免费的，但是您会有后续支持的费用。）

总之，大多数 IT 系统由现成的产品和资源以及定制的元素组成。确保所有的元素一起工作，同时满足系统的需求，这显然是架构师的责任。

9.7.3 采购产品

当您决定需要采购哪些组件之后，就可能需要遵循某些流程以确保按正确的方式选择这些产品，正如我们在补充内容“指导：产品选取”中所述。

如果您已经采用了基于重用的分层策略，那么，您想要获得的可重用资源将和架构分层一致。特别地，您可以把下列内容分开：

- 业务相关的元素，例如 CRM 系统或企业资源规划（ERP）系统。
- 独立于业务的元素，例如安全库（如轻量级目录访问协议服务器和公钥基础结构软件）、系统管理软件、关系型数据库管理系统、应用服务器、事务处理监视器和系统软件（如操作系统）。

指导：产品选取

以下是选取产品的一个简单的技术：

1. 回顾产品选取的标准。大多数组织除了产品选取标准之外还拥有一个优先供应商列表。如果存在一个企业架构，这种架构几乎肯定包含指导产品选取的政策。您应该复审这些标准以确保符合这些规则和政策。

2. 确认用于评估备选软件的标准。一些标准认为是强制性的（如果不遵循就会导致失败），而另一些标准仅仅是希望满足的。

3. 决定如何按这些标准进行打分，标准可能是二元的（备选产品符合或不符合标准），或是一个固定值（从几个选项中选取，例如“很好”、“好”、“一般”、“差”），或一个分值的范围（例如 0 和 10 之间的任何值）。还要考虑是否对一些标准使用权重，因为当您在判断产品对于所开发系统的适宜性时，不是每项因素都同等重要。

4. 确认可能满足所定义评估标准的备选产品列表。很显然，这一步需要很多的调查工作，包括要求供应商演示他们的产品和访问正在应用这些产品的网站（参考网站）。

5. 通过测评每个备选产品并根据标准记录结果来进行实际评估。在某些环境中，可能需要创建原型来比较竞争供应商产品的优缺点。这些原型可以由架构团队构建，或由供应商自己构建。

9.7.4 适应特定技术的模式

在转入特定的技术时，您要获得的一个很好的技术资源是一组特定技术的模式。很多供应商提供描述如何使用他们产品的模式，这些模式可能包含基于开放标准的元素。IBM（IBM 2009）、Sun（Sun 2009）和 Microsoft（Microsoft 2009）都为此在他们的网站上提供了建议和指导。这些模式可用于查看如何实现特定的组件。例如，Sun 创建了一组演示如何应用 Java EE 结构的 Java EE 蓝图（Sun 2009）。

接下来的例子使用 Java EE 来突出某些规范。我们在补充内容“Java 企业级平台概览”中提供了一个 Java EE 的概要说明。

Java 企业级平台概览

图 9.6 概要描述了一些用于典型的 Java EE 应用的关键技术，按架构层排列。

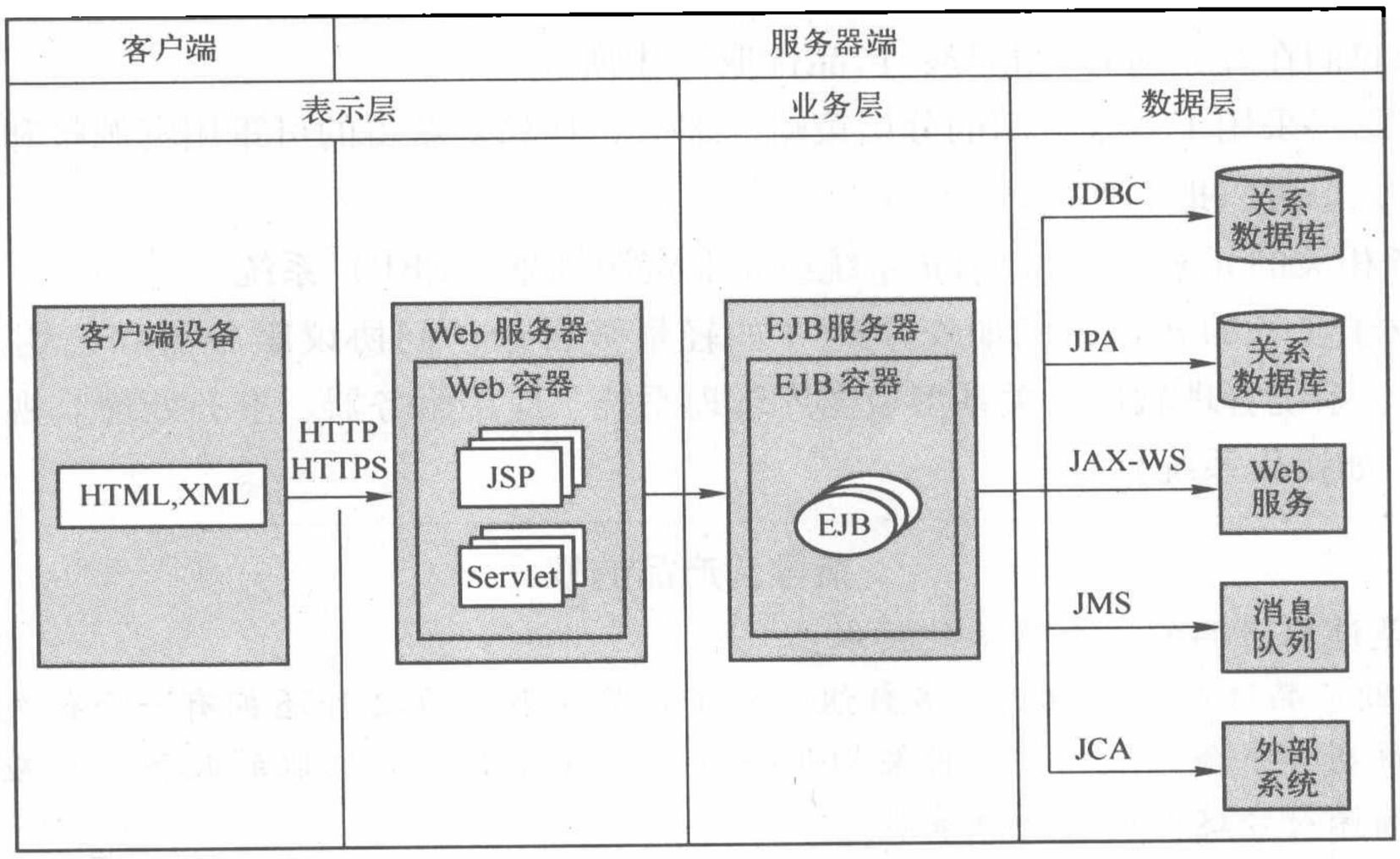


图 9.6 Java EE 概览

这张图包含以下元素：

- **一种标记语言**（例如 HTML 或 XML）。标记语言在客户端设备显示（如一个 Web 浏览器）。
- **Java 服务器小程序**（Servlets）。一个 Servlet 定义了如何处理来自客户端的一个请求及如何进行响应。Servlet 运行于 Web 容器中（稍后我们会进行介绍）。
- **Java 服务器页面**（JSPs）。类似于 servlet，一个 JSP 是一个描述如何处理请求和产生响应的文本文件。JSP 提供了需要生成标记语言语句时替代 servlet 的方法。JSP 在 web 容器中执行。JavaServer Face（JSF）框架是建立在 JSP 技术上的。

- **Enterprise JavaBeans**（EJBs）。一个 EJB 负责实现一个 Java EE 应用的业务逻辑的某些方面，并在需要的时候存储持久数据。EJB 运行在 EJB 容器中。
- **Java 数据库连接**（JDBC）。JDBC 为程序提供了访问关系型数据库的途径。
- **Java Persistence API**（JPA）。JPA 通过对象到关系（object-to-relational）的映射来支持数据的持久，它允许把持久的对象作为普通的 Java 对象（PoJos）来对待。这是通过声明对象持久属性的 Java 注释实现的，实际上提供了一个对象属性和一个数据库表字段之间的映射。注释是声明平台期望通过运行时注入来增加一些功能的元数据。
- **Java API for XML Web Services**（JAX-WS）。JAX-WS 提供了基于 Web 服务的 SOAP/XML 的支持。
- **Java 消息服务**（JMS）。JMS 提供了可靠的异步消息实现（如 IBM 的 WebSphere MQ）的接口。
- **Java EE Connector Architecture**（JCA）。企业应用的共同需求之一是连接外部资源。其中的一些资源可能是通过供应商特有协议访问的外部系统。JCA 提供了一个提供资源适配器（通常称为连接器）的标准方式。

容　器

J2EE 平台的核心是容器的概念。容器为在其中运行的应用组件（例如 JSPs、Servlets 和 EJBs）提供运行期支持。例如，EJB 容器为在其中运行的 EJB 组件提供组件生命周期管理（容器根据需要创建和删除应用组件）、事务管理、安全和持久支持。

EJB 接口

客户端对 EJB 的访问通过接口来支持。客户端永远都不会和 EJB 实现直接交互，因为 Java EE 平台把接口的方法映射到 EJB 实现中对应的方法。EJB 接口可以声明为本地（co-located）访问或远程（distributed）访问。

会话、实体和消息驱动的 EJBs

存在三种类型的 EJBs：会话（session）bean、实体（entity）beans 和消息驱动（message-driven）beans。

会话 beans，正如这个名字所示，是其状态在用户会话的上下文中有效的 beans。Java EE 平台指定了两种类型的会话 beans：无状态的会话 beans 和有状态的会话 beans。无状态会话 beans 是非常轻量级的，因为它不维护任何会话状态。无状态会话 beans 常用作协调其他 EJBs 之间的一系列交互而实际上并不维护它们状态的控制器。在另一方面，一个有状态的会话 bean 维护用户会话上下文中一个调用到下一个调用之间内存中的状态。

实体 beans 代表用于管理长生命期的粗粒度元素，因此，它们被提供长久支持。实体 beans 的例子有客户、订单和产品。一个实体 bean 可以使用 JDBC API，但如果需要平台管理对象到关系的映射，也可以使用 EJB3/JPA APIs。

除了会话 beans 和实体 beans，Java EE 平台还说明了消息驱动 beans。消息驱动 beans 被设计用于支持异步通信。发送消息给消息驱动 beans 的客户端在发送消息之后不需要等待响应。

对于所有类型的 EJB，都可以提供注释（annotations）使 EJB 容器正确地管理它们。另外，注释允许自动产生对 EJB 的 Web 服务的封装。

除了在各种可用的模式之间进行选取，您还需要根据选取的技术决定如何把这些逻辑架构中的元素映射到物理架构元素。下面是基于选取的 Java EE 平台的一些简单的规则：

- 代表一个用户界面的边界组件使用 JSPs 和 Servlets 实现。
- 连接外部系统的边界组件使用会话 EJB 和 JMS 实现。
- 控制组件使用会话 EJB 实现。
- 实体组件使用实体 EJB 实现。

把这些信息归纳起来，图 9.7 通过一个 YourTour 旅游线路预定控制器（Book Tour Controller）逻辑组件呈现了一个特定技术模式的例子。在这个图中，旅游线路预定控制器提供和需要各种接口。这个逻辑架构推断旅游线路预定控制器是无状态的，因为发送给它的任何请求都会简单地传递给适当的接口。

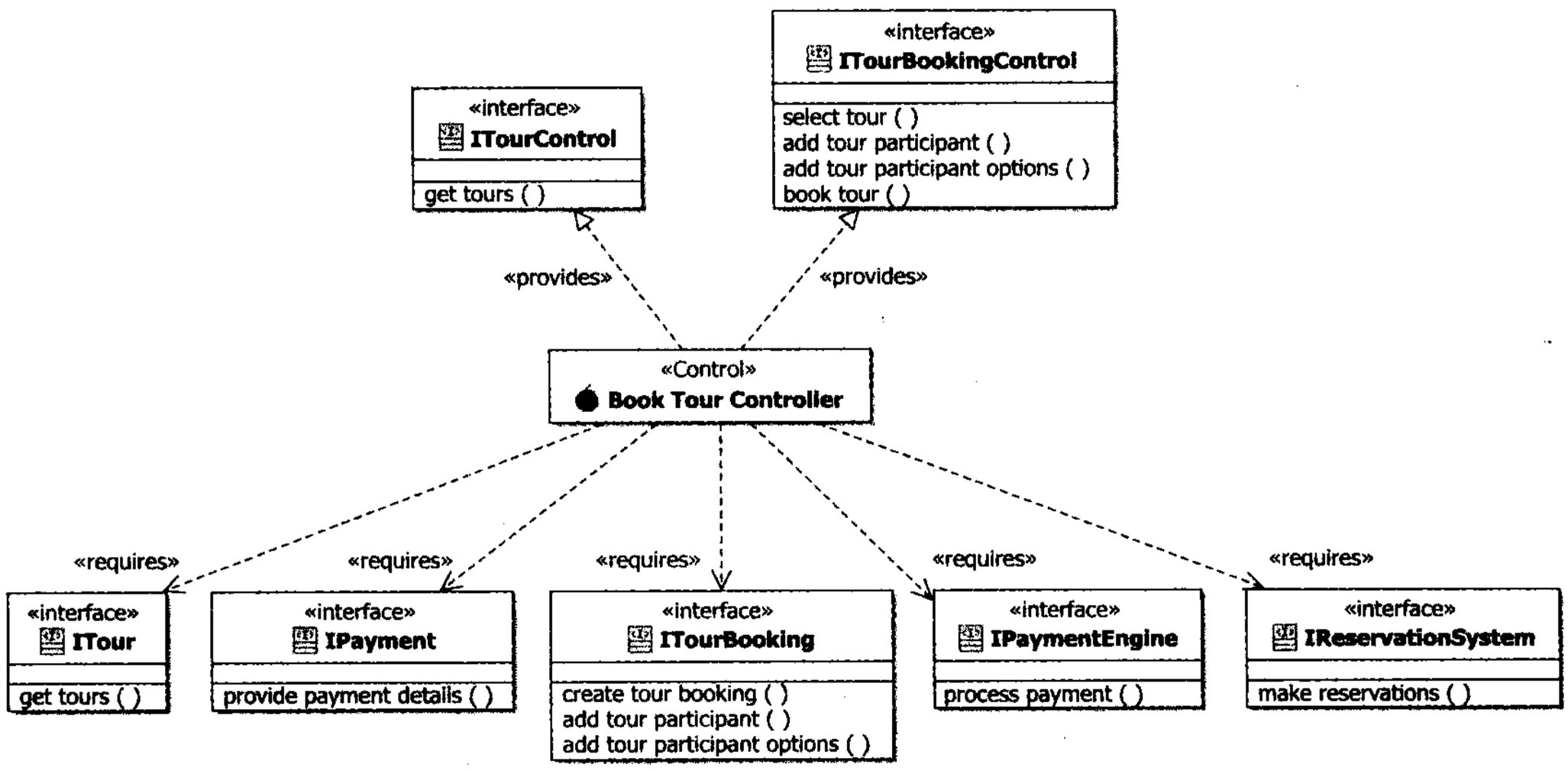


图 9.7　预定旅游线路的组件说明图

在从逻辑架构转入物理架构的时候，我们选择 Sun Java EE 的 session façade 模式。Session façade 模式“定义了一个较高层级的业务组件，它包含并集中了较低层级的业务组件之间复杂的交互”（Sun 2009）。图 9.8 显示了这个模式的结构。

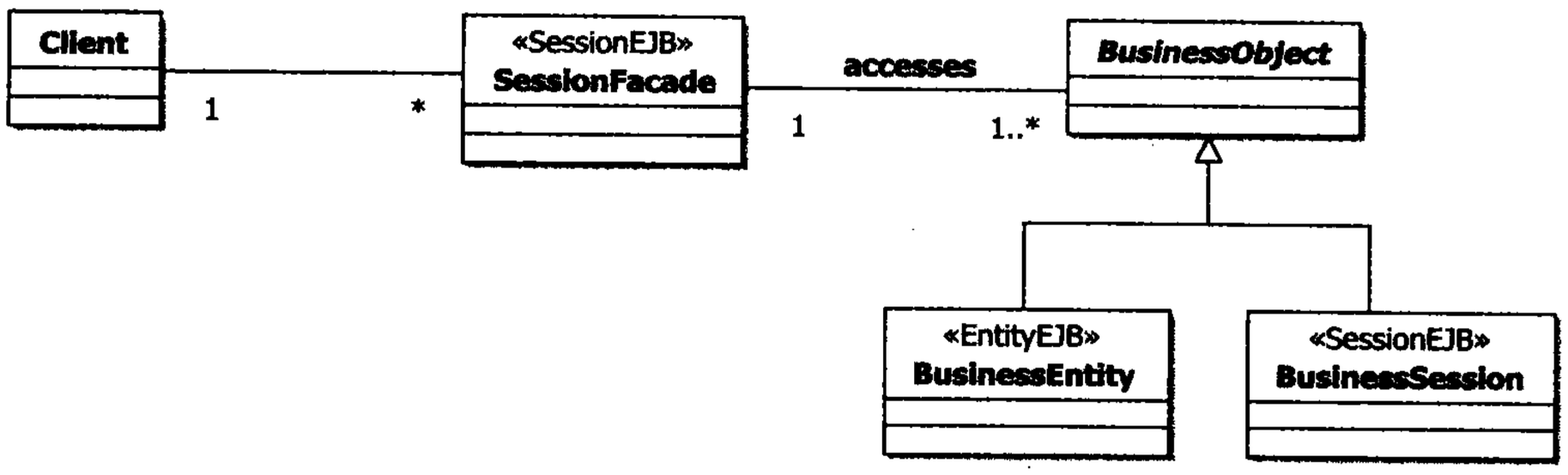


图 9.8　Java EE 的 Session Facade 模式

正如您所见，Session façade 被实现为一个会话 EJB 并管理一个或多个 BusinessObject 之间的关系。每个 BusinessObject 都实例化为一个会话 bean 或实体 bean。当实现这个 session façade 模式时，您需要做出好几个架构决策，如下所示：

- session façade 应该是有状态还是无状态的会话 bean？这个决策依赖于 session façade 提供的接口。只需要一个方法调用来完成一个特定服务的场景是非会话性的，因此能够使用无状态会话 bean 来实现。但是，如果这个场景需要多个方法调用来完成，那这个服务就是会话性的，这意味着有一些状态必须在每个方法调用之间保存。在这个场景中，将使用一个有状态会话 bean 来实现 session façade。
- 业务对象如何实现？这一模式所给的选择是业务对象可以是会话 bean 或实体 bean。会话 bean 通常提供一个不维护任何持久数据的业务服务，而实体 bean 却维护持久数据。

图 9.9 显示了如何使用 session façade 模式实现旅游线路预定控制器（Book Tour Controller）组件。BookTourController façade 为访问所有和旅游线路预定有关的信息提供了单一的接口，包括旅游线路参与者、他们的首选选项和关于预定的支付信息。这个 façade 是一个无状态的会话 bean，因为它不维护方法调用之间的状态。这个 session façade 封装了好几个会话 EJB，在这个模式中都用作业务对象：

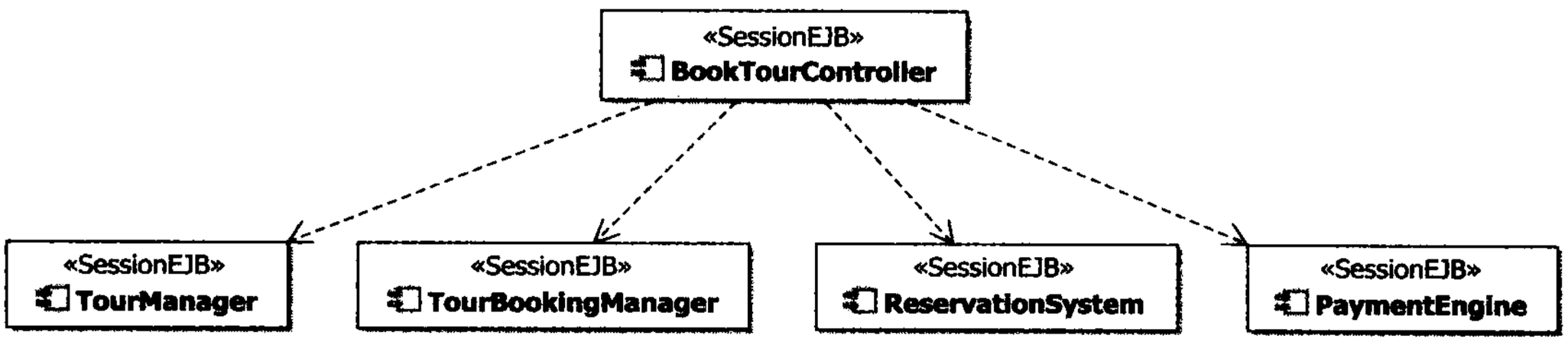


图 9.9　实现为一个 Session Facade 的旅游线路预定管理器

- 一个 TourManager 会话 EJB，用于管理旅游线路信息。
- 一个 TourBookingManager 会话 EJB，用于管理旅游线路预定信息。
- 一个 ReservationSystem 会话 EJB，用作和预定系统的接口。
- 一个 PaymentEngine 会话 EJB，用作和支付引擎的接口。

为了更完整地描述物理元素的特点，您可以基于物理元素创建一组需求实现来确保把相关的职责分配给它们。正如我们在第8章中所述，您通常通过创建一个或多个显示组件之间如何交互来实现需求的时序图来实现这个目标。如果想简要了解为物理架构创建这种类型的时序图的优点，请看补充内容“概念：物理架构中的需求实现”。

概念：物理架构中的需求实现

在第8章中我们介绍了需求实现的概念。这个概念允许您清晰地表达如何从解决方案元素的角度实现一个指定的需求（功能性的或非功能性的）。在那一章中，我们演示了如何从逻辑元素的角度实现需求。您可以使用同样的方法演示如何从物理元素的角度实现需求。

然而，在物理架构中执行这项任务到什么程度取决于好几个因素。如果您决定不创建逻辑架构（由于第8章中所描述的各种原因），您将无法根据物理元素追溯到产生它们的逻辑元素，进而追溯到它们解决的需求，从而没有办法通过这种追溯关系显示任何实现的需求。在这种情况下，您可以选择比较广泛的方式来分析物理元素如何实现需求。反之，如果您已经广泛地分析如何通过逻辑元素实现需求，您就不需要明确地表示如何使用物理元素实现这些需求，因为我们可以通过追溯物理元素到实现这些需求的逻辑元素来获得这些暗示。

最常用的方法介于两者之间：您从逻辑元素的角度实现一组关键的需求，也选择性地从物理元素的角度显示了如何实现某些特定的需求。

图9.10演示了一个包含物理架构中实现预定旅游线路用例主要流程的组件的时序图。在创建这种图的时候，我们同时也定义目标EJB的操作。另外，我们可能已经创建了一个类似的用以展示实现用户界面元素交互方面的用例事件流的图，如JSP页。

在图9.10中：

- 客户端调用session façade BookTourController来获得一个旅游线路列表。BookTourController调用TourManager EJB来获取一个旅游线路列表。
- 客户端调用BookTourController来选取一个旅游线路。BookTourController调用TourBookingManager EJB来创建一个新的旅游线路预定。
- 客户端调用BookTourController来添加一个旅游线路参与者。BookTourController调用TourBookingManager EJB在一个预定中添加一个旅游线路参与者。
- 客户端调用BookTourController来添加旅游线路参与者的详细信息。BookTourController调用TourBookingManager EJB为每位旅游线路参与者添加选项。
- 客户端调用BookTourController来预定这个旅游线路。BookTourController调用TourBookingManager会话EJB来保存每个旅游线路预定的详细支付信息。然后，BookTourController调用PaymentEngine会话EJB，通过支付引擎付款。最后，BookTourController调用ReservationSystem会话EJB，通过预定系统进行预定。

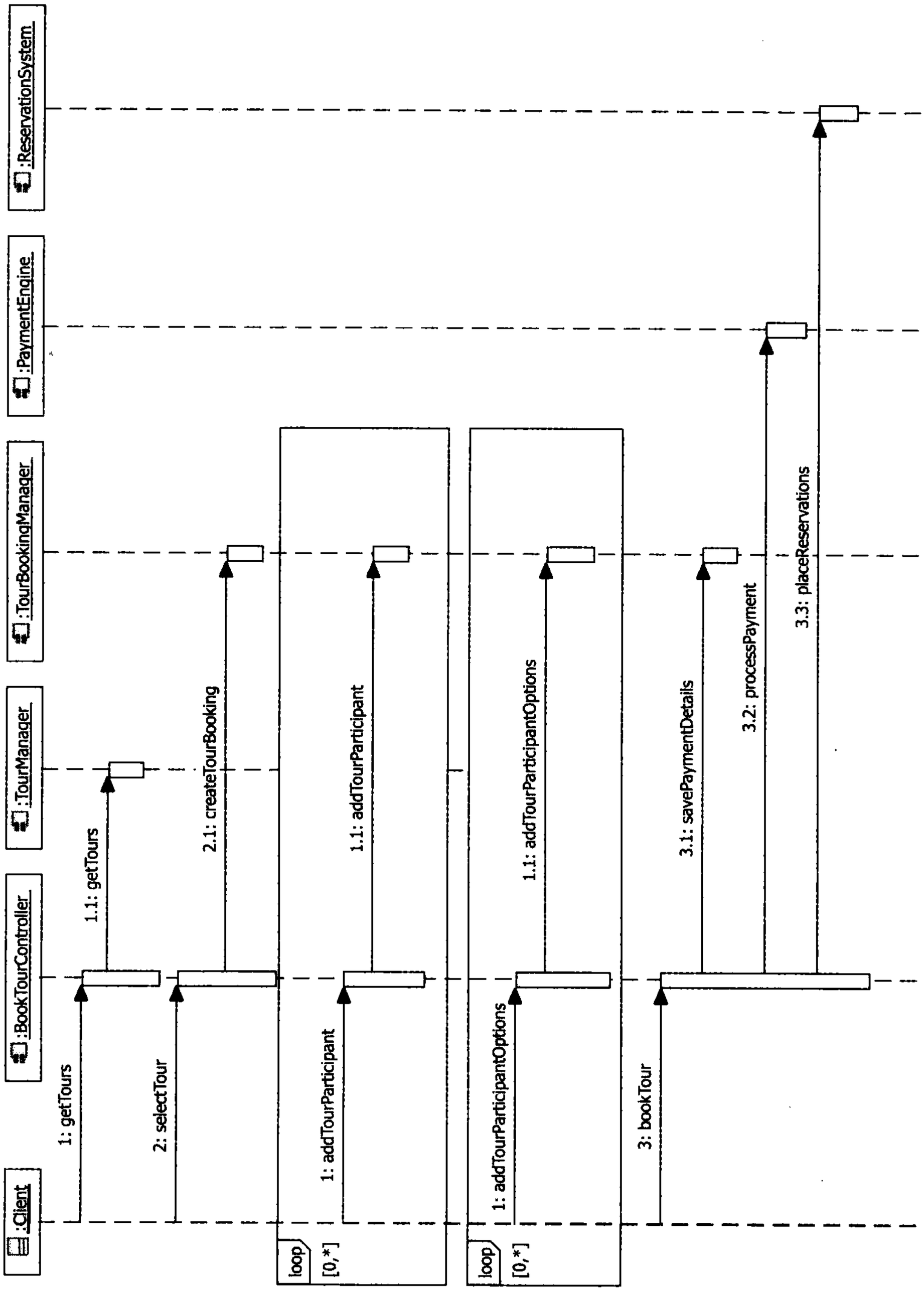


图9.10 预定旅游线路用例的主流程中的参与者

9.8 任务：概述部署元素

这项任务的焦点是从技术、产品和定制元素的角度确认物理部署元素。这项任务涉及以下方面：

- 满足所需服务级别特征的整体硬件配置，例如失败恢复、可扩展性和安全。
- 所需硬件的说明，例如处理器（基于它们的速度）、磁盘配置及网络带宽和延迟。
- 所需软件的说明（如操作系统），连同要求的版本和详细配置。
- 计算机硬件和软件以及连接它们的网络的选取。

9.8.1 映射逻辑部署元素到物理部署元素

把逻辑地点映射到物理地点就是一个确认解决方案中的物理地点的数量的例子。例如，一个“分公司”逻辑地点可以映射到“纽约分公司”和“东京分公司”这些物理地点。然而，在逻辑节点和物理节点之间不会一直存在这种直接的一一对应关系。就功能性元素来说，从逻辑架构到物理架构的转变可能会导致节点的分割或合并，这有多种原因：

- 一对一。如果单个物理节点就能够完全满足一个逻辑节点的要求，那就可以用单个物理节点实现这个逻辑节点。
- 一对多。为了满足性能、可靠性或其他的非功能需求，一个逻辑节点可能需要分割为多个物理节点。显然，这不仅会影响部署元素，也会影响功能元素（它们自己可能也需要分割以适应物理分布）。
- 多对一。逻辑节点最终可能是物理节点的一部分，而不是单独的处理单元；一个物理节点可能满足多个逻辑节点的需要。将多个逻辑节点合并为一个物理节点可能也是必需的，例如选择了一个必须部署在单个物理节点（例如为了提供更紧密的耦合，且为了提高性能和便于管理）上的封装的应用程序（packaged application），而它的功能在最初架构设计中位于多个分离的节点。

9.8.2 确认物理部署元素

考虑前面部分讨论的常规指导，您可以开发如下所示的一些物理部署元素：

1. 为每一个逻辑地点定义物理实例的数量和它们的地理位置。

2. 为考虑到的分配到逻辑位置的每个节点创建一个相应的物理节点（留意是否需要分割或合并节点，正如我们在前面部分讨论的）。

3. 为每个节点标记相应的特定硬件产品（假设这些节点是通过采购而不是构建所得到的）。如果不存在明确的对应产品，简单地列出备选产品，根据需求（尤其是非功能性需求）确认优先的选项。这可能会导致多个节点位于同一个地点，其中每个节点代表一个备选的硬件产品。

4. 通过复审非功能性需求，根据所要求的负载流量、可靠性、可扩展性等，设计物理节

点的规模，从而产生规模适当的平台（也就是说，有适当的实例、磁盘空间、处理器速度等的平台）。在一个大型的分布式系统中，这项任务本身就十分重要，可能需要详细的性能建模，进一步创建一个支持规模评估的**架构概念论证**可能会有所帮助。

图 9.11 展示了经过以上的步骤应该期望的详细程度。图中显示了 YourTour 系统的总公司（Central Office）地点实际位于伦敦，同时显示了服务器、工作站和其他设备以及它们的物理属性。尤其是，您会看到加入的安装了磁盘阵列的用于存储 YourTour 的所有持久数据的数据库服务器。

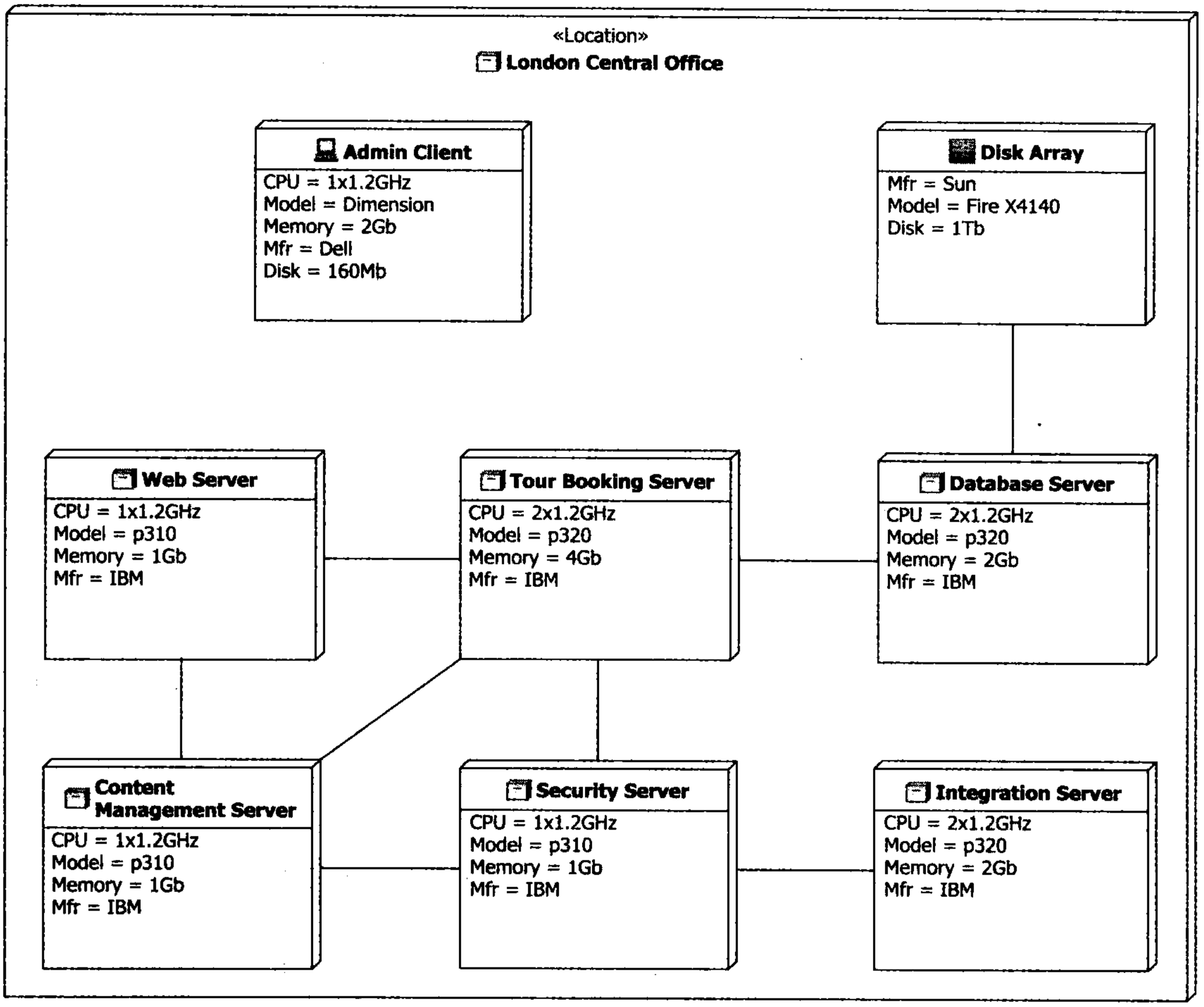


图 9.11　物理部署模型中的总办公室位置

图 9.11 还展示了节点之间可能需要的连接。在**细化部署元素**（Detail Deployment Elements）任务中把这些物理组件部署到这些节点时，如果需要，我们会确认和更新这些连接。其中没有显示来自管理客户端（Admin Client）的连接，因为它和图中所有节点之间都存在连接（除了磁盘阵列，它是一个设备）。

我们也可以在统一建模语言（UML）模型中通过为地点和它们包含的节点创建一个聚合关系来表示每个地点必须包含的节点的数量。这种关系可以显示它们之间关系的多样性。图9.12展示了伦敦总公司（Central Office）地点包含5个管理客户端节点。这种信息对于硬件采购无疑很重要——它是下一部分的主题。

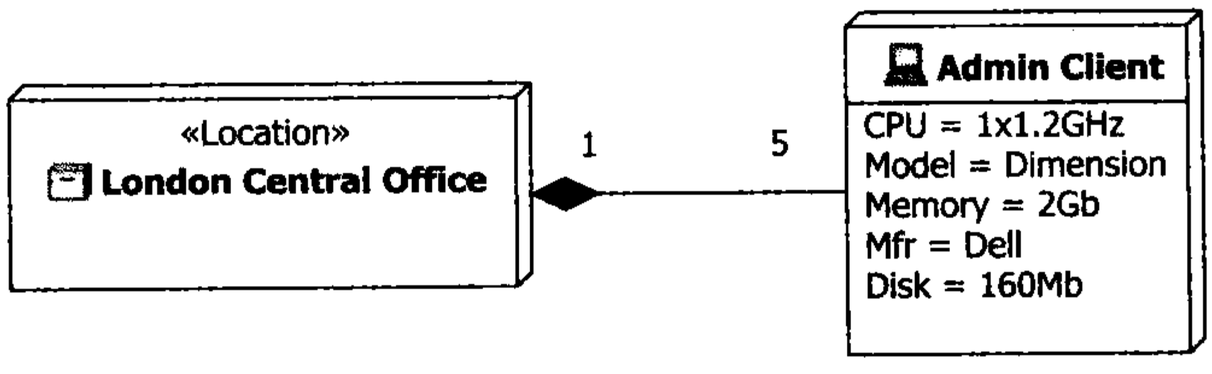


图9.12 显示特定位置的节点数量

9.8.3 采购硬件

逻辑部署模型（Deployment Model）是实现正在开发的系统所必需的部署元素的规格说明。和功能性元素不同，用于实现系统部署问题的各项元素很可能需要购买而不是构建。除非正在开发的系统非常专业，不使用现成的硬件实现物理架构中描述的物理节点，但这种可能性非常小。

决定了需要购买（采购）什么硬件之后，您可能需要遵循一些采购流程以确保按正确的方式选择产品，正如我们在这一章前面的补充内容“指导：产品选取”所述。这个指导主要是关于选取硬件的时机以及支持相同硬件规格说明的许多选项。

9.9 任务：检验架构

正如我们先前在第8章“创建逻辑架构”中考虑逻辑架构时所述，**功能性模型**（Functional Model）和**部署模型**（Deployment Model）通常由不同的角色开发，这两种模型难免在某些程度上不一致。这项任务的目的是确保架构在这些工作产品中的定义保持一致并一起满足所有的需求。

正如我们在第8章“创建逻辑架构”中所述，架构检验可以帮助您回答这样的问题：“我在正确地构建产品吗?”换句话说，您遵循了这个方法并符合所有的开发标准吗？这项任务的第二个目的是确保所有的架构决策都映射到架构模型中，这些架构决策记录在**架构决策**（Architecture Decisions）工作产品中并根据**架构概念证明**（下面讨论）产生的数据进行量化。

9.10 任务：构建架构概念证明

当您决定在架构中应用一种特定的技术之后，通过一个**架构概念证明**（Architecture Proof-of-Concept）来证明架构常常是有意义的。以后您可以使用这些从架构概念证明中收集到的数据。补充内容“检查列表：何时创建架构概念证明”给出了一些何时需要它的例子。

检查列表：何时创建架构概念证明

当出现下面的情况时，您应该考虑创建一个架构概念证明：

- 有一些功能性需求仅得到部分解决并需要进一步论证吗？
- 性能或要求的其他系统质量有可能出现问题吗？需要详细的度量数据证明系统满足这些质量吗？
- 存在高风险的或需要进行高风险的操作从而需要证明的组件吗？
- 现在使用的产品需要创建和其他产品或现存系统的接口且这些接口需要证明吗？
- 现在使用的产品是否有一个您需要显示给最终用户以证明能够满足这些用户需要的界面？
- 现在使用的产品或程序包需要复杂的定制或配置吗？（一些产品能够按多种方式进行配置，这会影响产品的性能或可靠性。）

架构概念证明的上下文和范围通常是通过复审不同的架构模型来决定的，这些模型强调上述列表的一个或多个方面。您还应该检查项目的 **RAID 日志**来帮助确认在哪些范围中存在技术风险，并确保概念证明关注处理那些必须解决和适用的范围。

9.11 任务：细化功能性元素

对于**概述功能性元素**（Outline Functional Elements）任务，每项技术都有它自己的特性，这会对功能性元素的接口和操作方面产生影响。例如，您懂得通过一个业务接口访问 EJB 的实现，而不是直接访问（Java EE 环境会在应用中注入合适的代码以使得这种访问成为可能）。

在这项任务中，当提供足够详细的信息让设计者和开发者构建系统的时候，您也要考虑这些特性。然后，物理架构就变成了设计者和开发者遵循的蓝图。这项任务的执行正是遵循逻辑架构中细化功能元素的描述，正如我们在第 8 章“创建逻辑架构”中所述。

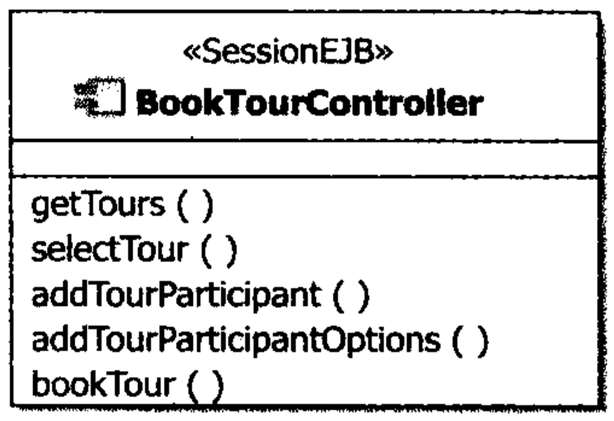


图 9.13 BookTourController 会话 EJB 的操作

在实际工作中，尤其是对于 Java EE 和 .NET 这样的标准技术，在提供足够详细的规范来支持详细设计或编码方面，只要把特定技术的模式和关注的元素简单关联就足够了。考虑图 9.13 中所示的 BookTourController，其中显示了基于它参与的所有用例流程的实现而分配给它的所有操作。

您知道 BookTourController 是无状态的会话 EJB。您可能还规定在模式中所有 EJB 的前端都有一个 JavaBean 类以提供对 EJB 的简化访问。比指定这个 JavaBean 类更好，您可以仅仅简单地声明一次这个模式并暗示这些元素，而无需在模型中清晰地表达它们。这个方法也可以应用于其他的元素，这种方法很有用，因为以后您可以使您的模型尽可能地简洁（不会到处都是不必要的细节）而不丢失任何有用的细节。

您可以通过提供操作签名及先决条件和后置条件来进一步添加细节。以这种方式在物理架构中细化操作签名确保组件接口的定义没有歧义。这意味着您可以放心地把这些组件交给开发

人员去实现并确信（假设这些接口符合需求）这些组件会按照期望的方式实现。

图9.14显示了BookTourController和作为其参数及返回类型的Java类。您可以从逻辑架构中找到每个操作需要传入的信息，也要在物理架构中指定这些元素时应用任何特定技术的考虑因素。

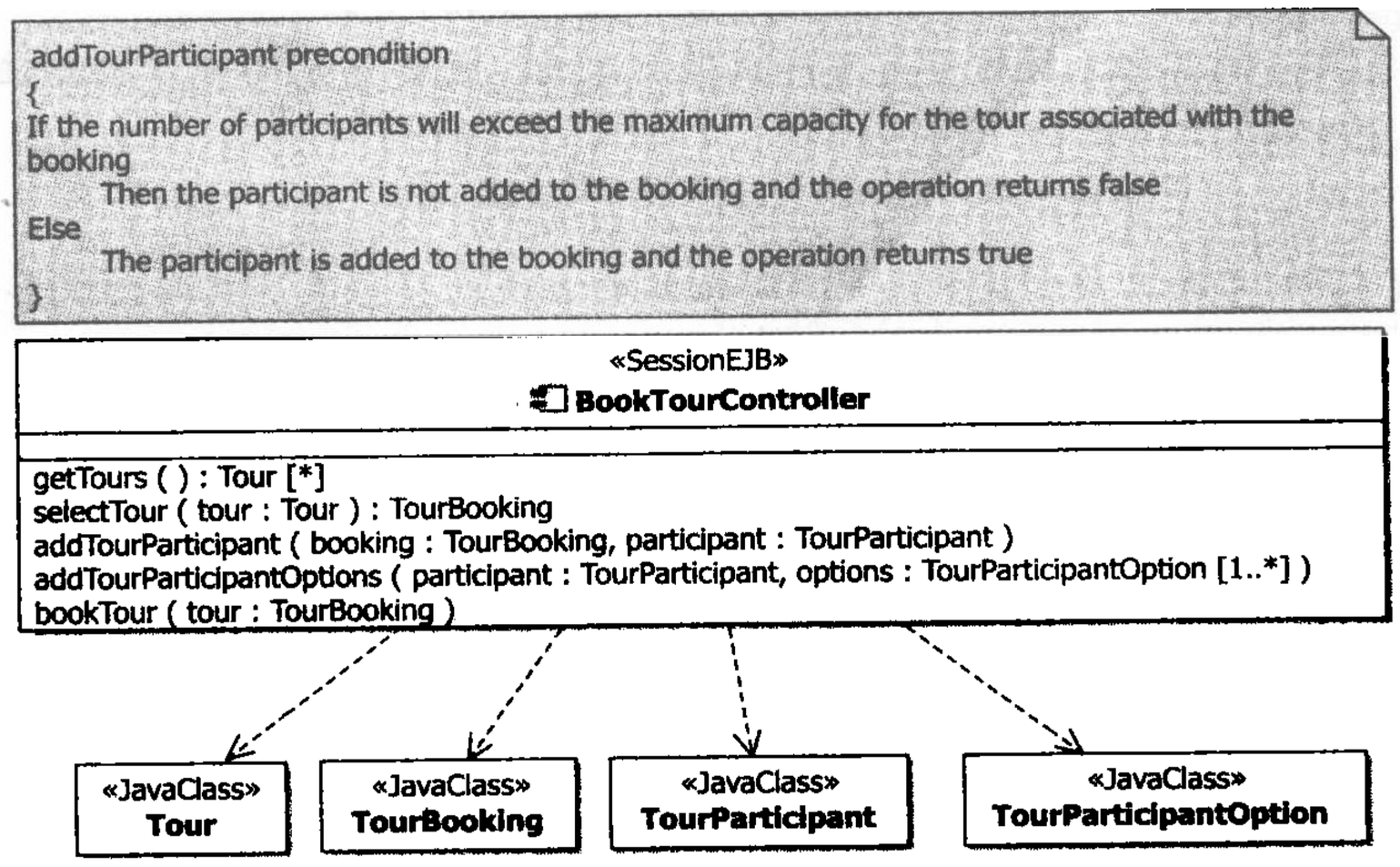


图9.14 当操作BookTourController会话EJB时的参数类型

正如您在逻辑架构中所做的，就是在**功能模型**（Functional Model）细化的时候需要把**功能模型**中的组件（和它们的接口）和**数据模型**（尽管在这时候数据模型按物理表的方式考虑）中的元素连接起来。显然您已经确定了需要持久的元素（在Java EE环境中，这些数据通过实体EJB和JPA对象管理）。图9.15显示了部分持久的元素，这个图使用一个UML概图描述一个关系数据库的结构（每个类代表一个数据库表，特定的属性规定为主键，外键通过依附于一个关联的角色名称来表示）。这个数据库结构反映了所有持久类的结构，受到逻辑架构中确定的业务类型的影响，当然也受到**业务实体模型**（Business Entity Model）的影响。也可以明确地显示一个数据库表和任何使用这个表的持久类之间的关系，尽管这个关系没有在图9.15中显示出来。

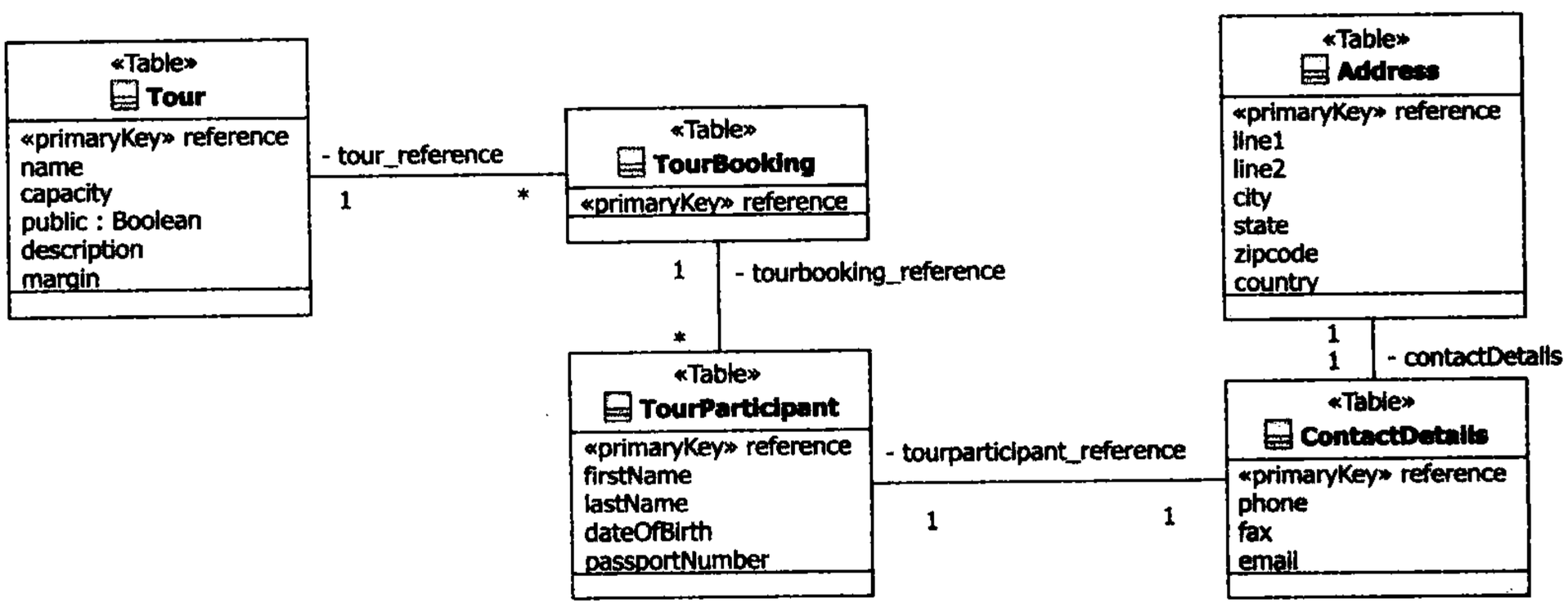


图9.15 摘录自物理数据模型

9.12 任务：细化部署元素

细化部署元素主要涉及确认那些连接物理组件和物理节点的部署单元，以及必须存在于节点之间用于支持物理组件之间交互的连接。

确认了这些物理组件并确定它们可以采购还是构建后，您可以考虑这些组件的物理部署地点。这项任务考虑以下内容：

- 对于定制的组件，决定如何对这些组件打包，以便可以通过确认合适的部署单元来把它们部署到相关的物理节点。
- 对于已经选取的产品，查看产品的文档并弄清楚这些产品如何和正在开发的系统一起部署。对于任何已经选取的可定制组件（如打包的应用），您还需要确保所有的配置选项和任何需要定制的地方都有详细说明。

表9.3列出了YourTour系统中一组物理组件到部署单元的映射。这些映射是逻辑架构中确定的部署单元的反映，但要考虑技术，也要考虑这些物理组件需要部署在不同的服务器这一实际情况，如图9.16所示。例如，JSP和servlet部署在一个WAR文件中，而EJB部署在Java EE平台上的一个JAR文件中。

表9.3 把组件映射到部署单元

组　　件	组件类型	部署单元（DU，Deployment Unit）	DU类型
TourManager	Entity EJB	tourBookingEntities. jar	JAR
TourBookingManager	Entity EJB	tourBookingEntities. jar	JAR
BookTourController	Session EJB	tourBookingControllers. jar	JAR
ReservationSystem	Session EJB	integrationComponents. jar	JAR
PaymentEngine	Session EJB	integrationComponents. jar	JAR
CRMSystem	Session EJB	integrationComponents. jar	JAR
TourBookerInterface	JSPs	tourBookingUIComponents. war	WAR

除了把组件分配给部署单元，您还可以把部署单元分配给相关的节点。图9.16描述了YourTour总部更新后的部署模型的一部分，显示了tourBookingUIComponents. war和tourBookingEntities. jar部署单元的位置。任何连接都添加或精炼为部署在每个节点上的组件之间交互的结果。

尽管类似于图9.16所示的图足够用于展示部署的概要信息，但是，在这样的图中您也只能获得这么多的详细信息。在大多数真实的场景中，您可能需要额外的支持信息，如表9.4所示。这种详细的节点物理说明信息显示了部署在它上面的所有组件，包括像中间件和操作系统这样的技术元素。

请记住，在这个阶段详细的物理设计还没有完成，架构师只是提供了指导来确保详细的物理设计能够在必需的约束条件（以确保架构的完整性）内完成。对于大型的、复杂的、分布式的系统（这类系统可能需要部署许多不同的元素），尽早地计划如何打包和部署系统尤其重要。在把软件打包成适当的部署单元时，您需要考虑以下问题：

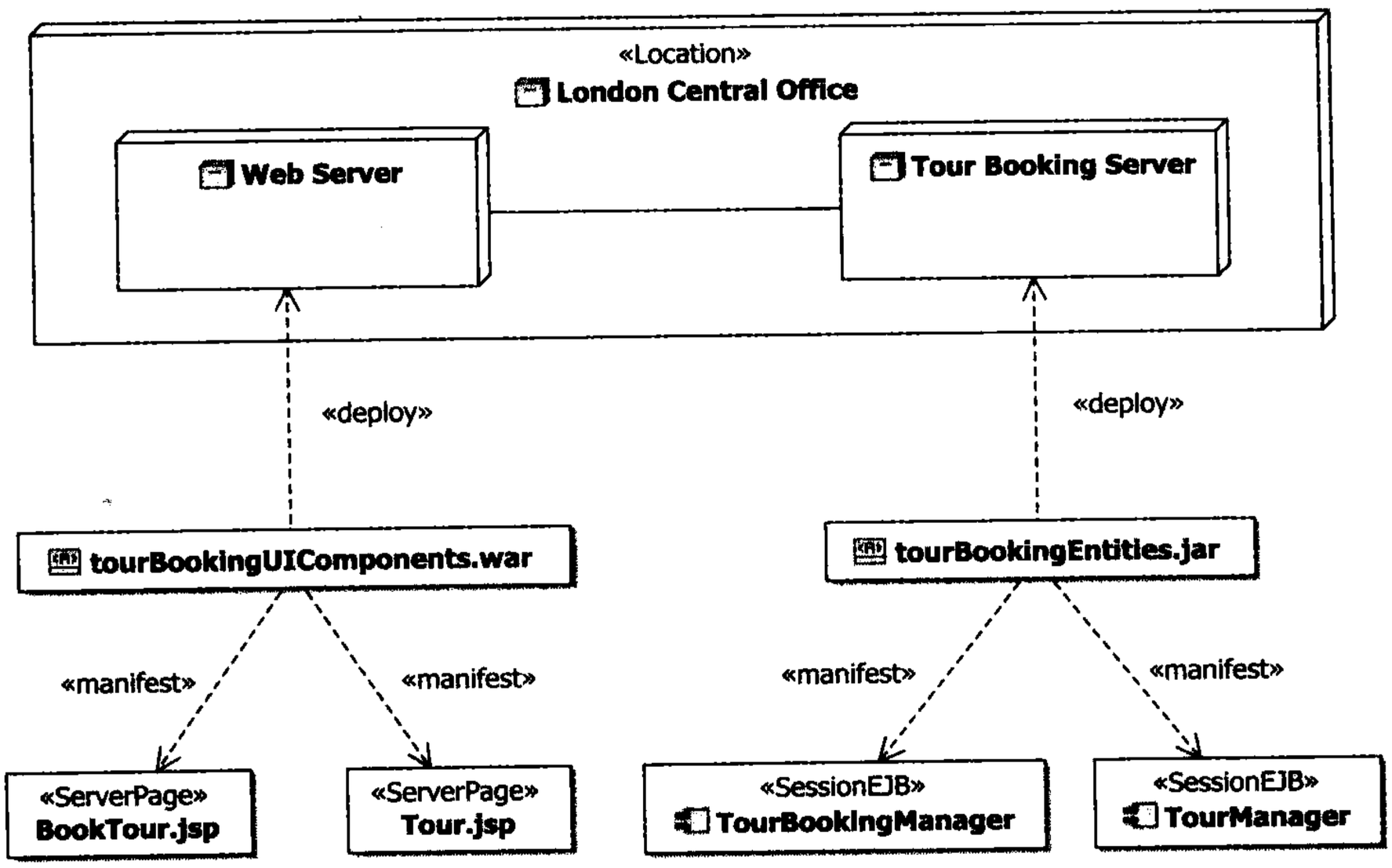


图 9.16 分配给伦敦总部服务器（子设备）的部分组件

- 任何存在于配置管理、变更控制过程和系统之间的联系。部署单元必须反映出所包含的各种组件和产品的正确版本。在一个相关的说明中，您应该考虑不同软件元素之间的依赖关系，特别是在需要一些来自不同供应商的常用库的地方。
- 软件发布和安装的物理机制（例如提供通过 Internet 访问，邮寄安装磁盘，或派人现场安装软件）。
- 考虑和国际化及特定语言环境有关的各种必要的数据和资源。
- 必须支持的各种测试环境和每个环境需要的软件配置。大多数系统需要进行单元测试、集成测试、系统测试和验收测试，每个系统都可能需要不同的环境来进行这些测试。如何把软件部署到这些环境中和如何在不同的环境间进行改善，这些都是架构师十分关心的问题。

表 9.4 旅游线路预订服务节点的组件映射

节点：旅游线路预订服务器	
CPU：2x2.1 GHz	
Model：p320	
内存：4 GB	
厂商：IBM	
组件	版本
TourManager	V 1.0.0
TourBookingManager	V 1.0.0
Solaris	V 9.0.0.3
IBM WebSphere Application Server	V 7.1.1

架构师还应该清楚软件部署之后必须支持的操作关注点，因为这些关注点会影响物理架构。架构师应该考虑以下问题：

- 为新系统填充数据的机制。架构师不仅需要关注如何将新数据输入系统（如用户、产品、顾客等），还需要关注如何集成被替代系统中的数据（如果存在这种情况）。数据

移植是指把现有系统中的数据装入新系统的过程。在大型的、复杂的开发中，这个过程本身就可能是一个完整的项目，它需要对现有的数据进行分析、对数据进行清理以消除错误和重复，还要研究移动这些现有数据的策略（例如，一种名为“大爆炸”的方法，这种方法是在将数据整个快速地移植到新系统。或者采用分阶段的方法，每次移动部分数据）。数据移植也可能会在架构中引入新的子系统和组件，如一个抽取、转换、装载（ETL）的子系统或文件传输组件。

- 对更新现有系统的时间的约束，特别是对于那些正在使用并需要全天候运行的系统。

在这项任务中，您还需要通过考虑一些要求的涉及性能、可靠性和可扩展性的服务级别的特性来精炼每种类型的物理节点的数量。例如，可能只有通过重复一些节点来获得高或持续的可靠性，这样如果一个节点失效，系统会切换到另一个节点。类似地，可扩展性需求可能要求使用多处理器（被称为垂直可扩展性）或多节点（被称为水平可扩展性），最终，这些考虑增加了部署节点的详细说明。

基于物理架构中定义的节点（和地点）之间的连接，您还可以决定如何使用现有的或新的网络设备实现这些连接，如局域网或广域网、路由器、网桥等。这项任务需要规定这些连接之间的流量以确保实现的新系统能保证足够的流量，这样网络不会成为瓶颈或对系统性能产生负面的影响。

9.13 任务：确认架构

确认物理架构是建立在逻辑架构的说明过程中进行的相同名称的任务之上的。它回答了这个问题“我使用已经选择的技术和产品构建的系统正确吗?”换句话说，系统满足声明的需求了吗？从而达到利益相关者的期望了吗？

进行确认的步骤和在定义逻辑架构中对应任务的步骤相同。但是，所要满足的标准却不相同，或者至少关注点不同。确认物理架构时所考虑的标准和确认逻辑架构的标准相似；附录 B 中提供了适当的检查列表。

当您确认架构时，需要依照需求核实目前已经创建的所有模型（就是**功能模型**、**部署模型**和**数据模型**）。另外，在确认的过程中还会用到**架构决策**（Architecture Decisions）工作产品中记录的关键决策，**架构概览**（Architecture Overview）工作产品也会用到。

9.14 任务：更新软件架构文档

软件架构文档（Software Architecture Document）会更新以反映已经定义的物理架构。正如我们在本书前面提到的，软件架构文档的目的是提供一个工具来总结所有和架构有关的信息并把它们汇集在一起，以便传达和复审架构。这个文档通常是引用的工作产品中可用信息的一部分。

在更新软件架构文档来包含物理架构时，您所面临的挑战是决定是否扩展任何现有的文档或把一个可能会很大的文档分解为好几个单独的文档，从而共同组成**软件架构文档**工作产品。

可以通过好几种方法完成这种分解。例如，您可以根据逻辑架构或物理架构进行分解，或根据架构观点进行分解。

将这些架构工作产品汇集为一个完整工作产品的主要原因是为了将当前完成的工作传达给团队成员，也为了能够让利益相关者复审以便他们可以签收。

关于利益相关者的签收，大多数具有一定规模的项目都有一个或多个核准继续点（AtP，authorization-to-proceed），需要通过它们才能获得下一步的资金和资源。为了满足 AtP 的输入，作为 AtP 一部分的需要复审的重要工作产品通常需要定义一个基线。这就是创建**软件架构文档**的主要原因之一，它作为架构在某一时间点上可以由利益相关者复审并取得一致意见的视图。

9.15 任务：和利益相关者复审架构

这项任务几乎和第 8 章“创建逻辑架构”中的一样，尽管它针对的是物理架构。在这项任务中，您首先需要定义物理架构工作产品的基线，然后和关键的利益相关者复审**软件架构文档**和辅助的架构工作产品。把所有发现的问题记录在**变更要求**中。当复审结束时，您要在**复审记录**中简要地记录复审结果，包括所有的行动项。

这种复审可以为质量把关并确保物理架构足够稳定来支持依赖于它的各项任务。因此，相关的各方面都参与复审很重要——例如包括**项目经理**、**测试人员**和**开发人员**。

9.16 总结

开发软件架构是一个迭代的过程。架构师必须不断地精炼、检验和确认架构以获得对正在开发的系统感兴趣的利益相关者的认同。此外，随着信息的收集和对需求更好地理解，架构师能够从架构的一个概念视角转向一个比较物理的视角。

在这一章中，我们向您展示了如何通过应用与逻辑架构相同的流程模式来精炼和详细说明逻辑架构中的元素，从而创建一个明确说明用于设计和构建系统的技术、产品、资源和定制元素的物理架构。

第 10 章将研究软件开发项目中的其他方面，包括在复杂环境中架构设计所面临的特殊挑战。

第 10 章

进　　阶

从第 7 ~9 章讨论的过程关注描述在典型的软件开发项目中执行的关键任务，强调的是架构师这个角色。这些描述的上下文是一个相对简单明了的旅游线路预定系统。

在这一章中，我们要查看软件开发项目中的其他方面，尤其是架构师参与的到目前为止我们还没有讨论的方面，如测试、配置管理、变更管理和开发环境。另外，尽管这项研究案例简化了各种架构挑战，例如，满足系统的质量要求，考虑集成和封装的应用程序，但是仍然需要考虑另外一些复杂问题。这些复杂的问题包括调节大型系统、工作必须进行协调的大型团队及分布式的开发团队——所有这些都要在这一章中进行考虑。

10.1　架构师和项目团队

作为项目的技术领导，架构师对软件开发项目的很多方面产生影响。我们在第 7 章中讨论架构师参与定义需求规范。在第 8 章和第 9 章中，我们关注架构师参与的核心架构设计任务。

这一节的目的是讨论软件开发项目的其他方面，尤其是架构师参与的其他规范，因此也讨论架构师在团队中的其他角色。

> **规范**是用来组织任务（定义一个主要“关注范围”和/或工作协作成果）的主要分类机制。(OpenUP 2008)

10.1.1　架构师和需求

我们在第 7 章中详细讨论了在需求方面架构师的角色。在那一章，您看到了架构师对于需求拥有多种责任，如图 10.1 所示。架构师所做的最大贡献是确保考虑到最具有技术性的系统用户（如系统管理员），从而确保能够捕获那些最具有技术性的需求（特别是非功能性需求，如质量和约束）并进行适当的细化，也有助于定义请求的优先级和最终需求的优先级。

10.1.2　架构师和开发

开发规范的目的是进行详细的设计任务、定义系统实现的组织结构、实现详细的设计元素、对实现进行单元测试、集成单个开发人员的工作及最终产生可运行的系统。

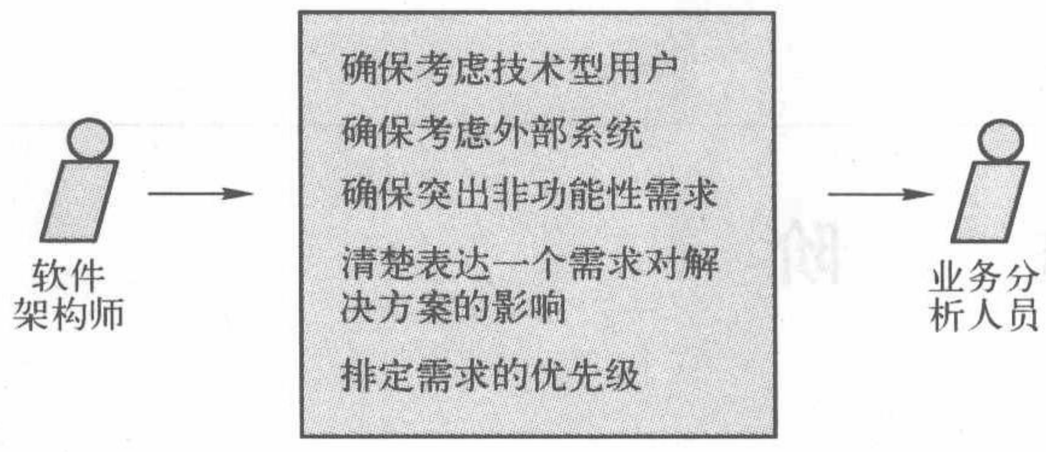


图 10.1 架构师关于需求的角色

显然，架构是详细设计活动的输入，因为它确认了系统的主要组件和它们的接口，正如我们在第 9 章中详细讨论的，还包括确认部署单元，如可执行工作产品。然而，架构师的影响不仅如此。如图 10.2 所示，架构师还负责系统实现中对架构有重大影响的方面，例如负责组织源代码的系统实现结构。一个结构良好的系统实现支持并行开发，因此，不同的系统元素可以由不同的开发人员（或开发团队）并行地实现。架构师还负责确认任何和开发有关的标准、指导和可重用资源。

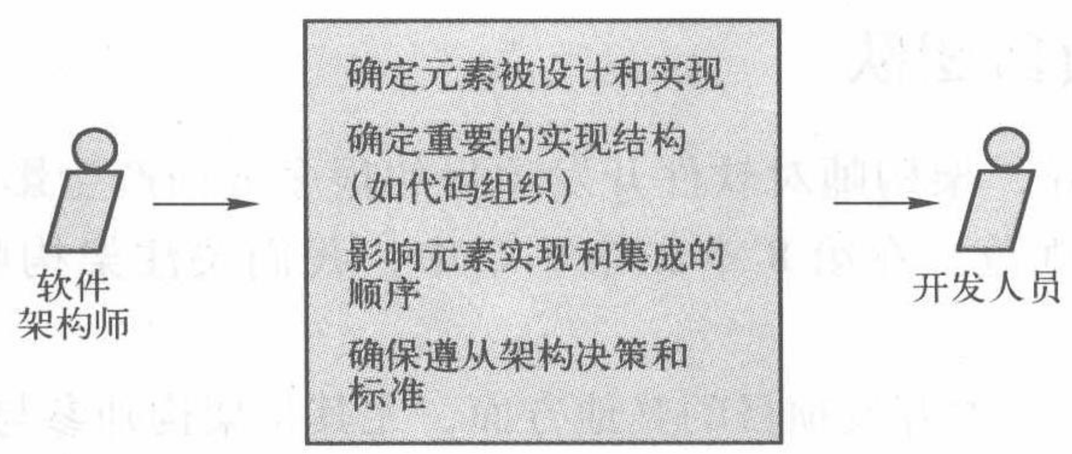


图 10.2 架构师关于开发的角色

架构师关于系统实现的另一个考虑是系统实现的不同组件按什么顺序集成以提供一个可交付系统。架构师还要对集成和测试提供优秀的指导。例如，把所创建的系统作为一连串可以单独集成和测试的部件，这应该是有可能的。尽管这些考虑看起来相当细小，但是，架构师还是应该要避免这样的细节风险，正如我们在后面的“缺陷：避免技术细节”部分中所述。

除了为某些和开发有关的技术问题提供特定的输入之外，架构师还总体负责技术成果，他们自己还要相应地使用这些成果。这种情况适用于所有的规范，尤其是开发规范。除了参与他们自己负责的领域外，架构师通常还会和项目团队成员一起紧密地工作以便指导他们。特别地，架构师需要确保所有的架构决策和标准都被遵循。

缺陷：避免技术细节

在第 2 章中，我们说，对于架构师而言，拥有技术知识、设计技能和一定水平的编码

技能很重要。尽管架构师可能主要关注架构的重要元素，但是，有时候他们必须参与详细设计和编码。

架构师们特别易犯的错误是总把这些细节交给其他人处理，不考虑这些细节。这种架构师会失去团队成员对他的尊重，容易带来坏名声并伴随许多负面问题，例如眼高手低、在象牙塔里远离项目，或只会夸夸其谈而不会动手。好的架构师应该亲自动手，在需要的时候深入到技术细节中。

10.1.3　架构师和测试

测试规范的目的是找到并记录软件质量中的缺陷，以便能够对发现的软件质量问题提出警告，还要核实系统功能是否符合设计和要求。完成开发规范中的单元测试任务后，测试规范关注开发人员提供的集成单元和整个系统。

测试规范特别关注**集成测试**、**系统测试**和**验收测试**。集成测试，正如它的名字所示，关注测试好几个开发人员的集成工作。系统测试关注整个系统的测试并确认系统所有的功能。验收测试通常是最后的测试活动，因为软件已经部署到生产环境中。验收测试的目的是验证最终用户可以使用软件来执行系统功能。

架构师在测试规范中的参与内容如图 10.3 所示。架构师确保架构是可测试的，是比较非正式地测试过的。从可测试性的角度来看，架构师确保系统可以在定义的功能性需求和非功能性需求方面进行评估。在非功能性需求方面，架构师必须确保运行期质量可以评估，取决于定义的约束，还要确保所有的约束都被评估。例如，如果需求声明系统和外部系统之间的所有通信都必须加密，则必须采用某种方法截获系统和外部系统之间的信息，以便能够评估这种加密性。

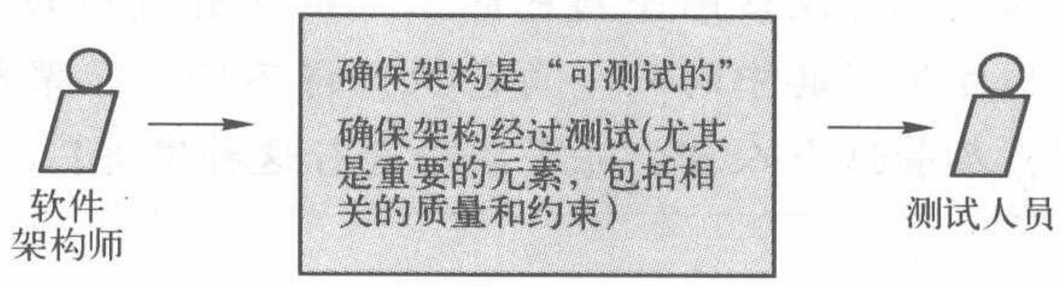


图 10.3　架构师关于测试的角色

另外，架构师最感兴趣的是确保正在开发的系统能够进行全面的测试，因为最终的系统质量主要是他们的责任，他们是项目的技术领导。因此，架构师们经常和测试团队一起工作以确保用比较具有挑战性的系统需求来引起大家对测试活动的关注。因为架构师有助于定义需求的优先级（从而有助于考虑某个迭代中的需求）和确认满足需求的解决方案元素，所以，架构师有利于在任何时候确认系统需要测试的范围。架构师还和测试团队一起判断是否存在优化一些测试工具（和测试数据）的机会。

10.1.4 架构师和项目管理

项目管理规范的目的是为管理项目提供一个框架，为制定项目计划、安排项目人员、执行项目和监管项目提供实用的指导，为管理风险提供一个框架。

正如第 2 章中所述，架构师在定义进度、工作分配、成本分析、风险管理、技术获取等方面提供计划支持，如图 10.4 所示。在第 2 章中，我们得出如下结论：架构之所以能够提供如此多的帮助是因为它确认了系统中的重要组件和它们之间的关系，进而适当地使用了这些信息。

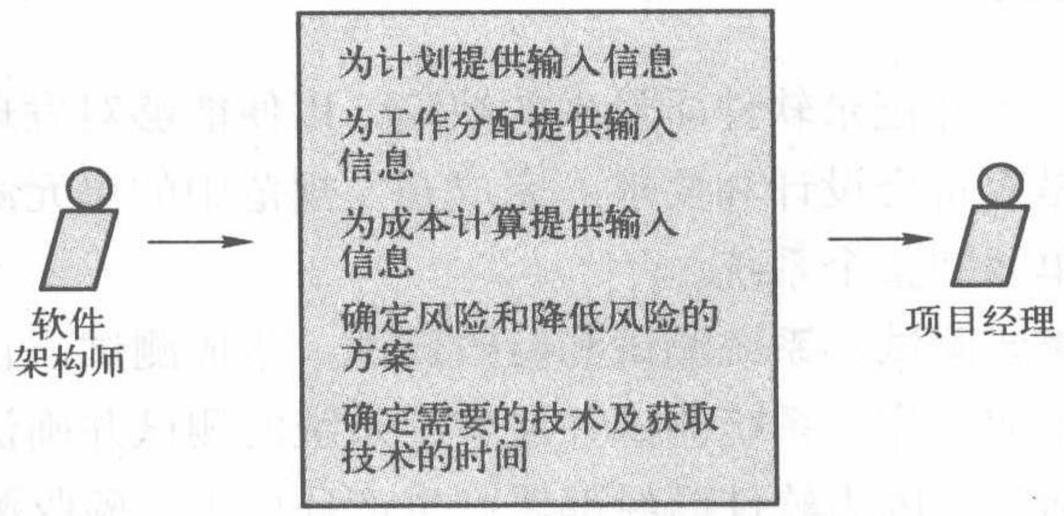


图 10.4 架构师关于项目管理的角色

尽管在本书描述的流程中人和角色是分离的（因此同一个人可以扮演多个角色），但是，一个特别的缺陷是同一个人同时扮演架构师和项目经理，正如后面的“缺陷：架构师和项目经理是同一个人”部分所述。

缺陷：架构师和项目经理是同一个人

架构师和**项目经理**的角色都很重要，而且通常需要花费很多时间。这两个角色的关注点差异也很大。由一个人同时扮演这两个角色的项目很少有成功的，因为一个人总会倾向于这一个或另一个角色。结果，其中的一个角色执行得不恰当。架构师和**项目经理**之间相互影响的好处不存在了，尽管这个人可以在头脑中进行这种思考！

10.1.5 架构师和配置管理

配置管理规范的目的是确认和管理应该置于配置管理控制之下的元素。许多元素必须进行配置管理：文档、模型、源代码、可执行代码、项目计划、测试脚本等。这些元素具有各自的版本，还分配给了一个或多个配置，其中一个配置定义了系统或部分系统的一个特定版本。元素可以是整个工作产品（如果单个文件代表整个工作产品）或部分工作产品（如果一个文件代表部分工作产品，如一个用例说明代表部分的**功能性需求**工作产品）。

配置管理策略中的一个必要元素是支持所有元素（因此，包括项目的所有工作产品）版本库的合适的工具。这个工具用于整个项目生命周期并包含元素的当前版本和历史版本。一个

好的配置工具还能够根据所持有的信息提供度量数据，例如元素的变更率（这能够表示稳定性，架构师可能对此特别感兴趣）。

版本库的结构通常既是项目定义的工作产品的反映，对于某些元素来说，也是已经定义的架构的反映（例如，当存储模型或代码的时候）。这种布局会让团队成员更容易地在版本控制系统中找到放置他们产品的位置。因此，架构师需要确保定义适当的结构，如图 10.5 所示。后面的“缺陷：配置管理忽略了架构”部分讨论了没有进行这种布局的后果。

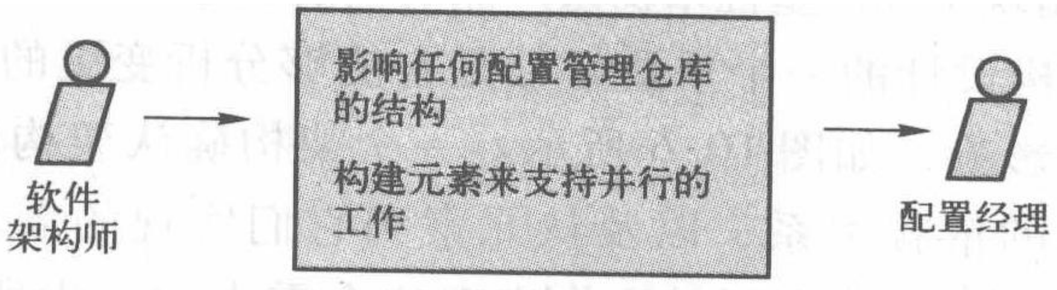


图 10.5　架构师关于配置管理的角色

结构定义还应该尽量减小使用者同时修改各种工作产品时需要的比较和合并。除了定义配置管理版本库中元素的结构，架构师还要帮助定义修改存储在版本库中元素的方法。例如，可能决定一个代码文件在某一时刻只允许一个人进行修改。在这种情况下，签出的文件会被锁住，这意味着其他人可以复制但不能修改它。如果您决定两个人或多个人可以同时修改同一个文件，这就需要使用者向版本库提交他或她的工作时，比较并合并这些改动。

配置管理同时还和构建 - 发布的机制密切相关。这种机制自动化了软件发布的编译、测试和打包。例如，每一步都会访问配置管理系统中的文件，最终根据各部分的正确版本生成可运行的系统。

缺陷：配置管理忽略了架构

如果开发团队没有给予配置管理策略足够的重视，就会使他们自己陷入真正的困境。当一个团队没有认识到架构和配置管理之间的关系时，就会发生这种情况。主要的症状是构建的产品没有反映出项目工作产品中已有的改动。

因为结构没有很好地定义，而把文件放在配置管理版本库中错误的位置时，也可能会发生这种情况。这样使用者会优先使用元素的旧版本（它们被放置在版本库中正确的位置）而不是最新版本（它们被放置在版本库中错误的位置）。

另一个原因是因为允许过多的使用者同时修改同一个文件，而不同的使用者未能正确地比较和合并所作的修改，从而造成文件中的修改丢失了。这种情况通常是由于没有充分地考虑以下情况而导致的：哪些文件存放哪些信息和使用者在工作时会如何访问这些文件。

这两项挑战都会受到架构师的影响，需要他能够充分考虑文件的存放和文件将接受的存取。架构师在确保一个简单的配置管理版本库结构准备就绪，并制定了相关政策后，特别希望工作能够尽可能地同时进行。

10.1.6 架构师和变更管理

另一个和配置管理密切相关的规范是变更管理。变更管理规范的目的是限制和审计工作产品的变更。特别地，变更管理规范包括对所需组织元素（例如一个变更控制委员会）的考虑以评估一个变更对成本、进度和现有系统的影响。它还能够对工作产品中的变更进行跟踪，这可以作为软件的变更历史和原因。最后，变更控制规范还关注根据项目开发过程中找到和修复的缺陷的类型、数量、出现率和严重性来衡量产品的当前状态。

根据这个范围，架构设计的一个重要好处是它能够分析变更的影响，允许架构师在变更发生之前推断产生的影响，如图 10.6 所示。一个架构确认架构的主要因素和它们之间的相互作用、元素之间的依赖关系，以及从元素到它们实现的需求之间的追溯关系。例如，我们可以分析需求中的一个变更对协作实现这个需求的元素所产生的影响。类似地，我们可以分析特定元素的变更对依赖它的其他元素产生的影响。这种分析对判断变更的成本和风险有很大的帮助。变更控制委员会（架构师参与其中）在定义变更的优先级时会使用这些信息。

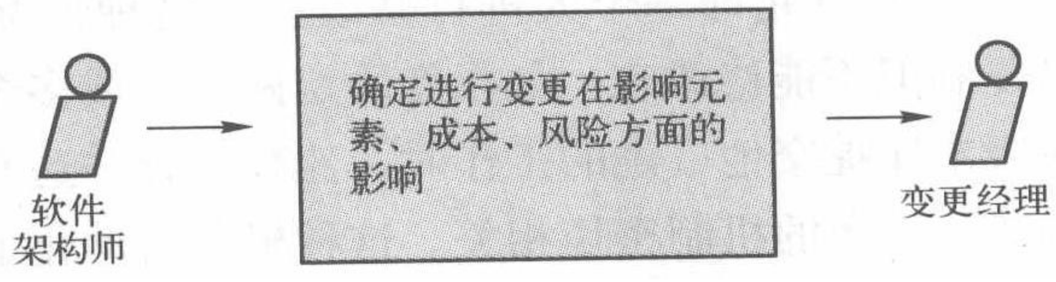


图 10.6 架构师关于变更管理的角色

10.1.7 架构师和开发环境

开发环境规范的目的是为开发团队提供软件开发环境。环境本身由几种元素组成，大的组织可能拥有一个部门或团队专门负责提供开发环境。这种团队甚至还可能拥有开发环境架构师的角色（Eeles 2008）。

开发环境的元素中包含以下内容：描述了开发团队的成员应该如何一起工作的一个方法或流程、在方法中实现自动化的工具、能够让团队成员掌握特定技术的资料，以及支持方法、工具和项目启动的基础设施。另外，开发环境可能包括支持环境的团体，如一个专门的支持部门，还包括项目如何采用这种环境的指导。

架构师和其中的几个元素有关，如图 10.7 所示。例如，架构师负责定义所有团队成员使用的架构标准和架构指导（包括任何编写架构文档的标准），这些也是方法的一部分。总体而言，标准是要必须遵守的（规定），从架构的角度来看，可能包含某些技术标准（如 Java EE 企业版标准）、开发标准（如设计或编码标准）等。另一方面，一项指导（guideline）对如何开展一个或多个活动（例如，如何从系统需求中找出影响架构的解决方案元素）提供了规范性指导（guidance）。

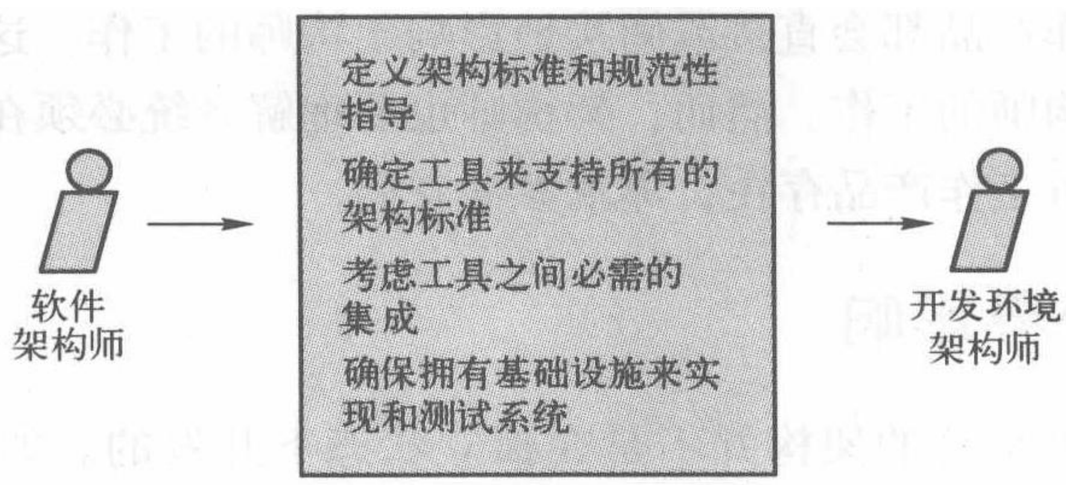


图 10.7　架构师关于开发环境的角色

架构师同时提供可选的工具，因为这些工具需要用于支持架构师要求的开发标准。例如，如果已经要求使用统一建模语言（UML）编写架构文档和详细设计，那就需要一个合适的 UML 建模工具。类似地，如果用于实现的技术基于 Java，那就需要合适的 Java 开发工具。另外，架构师要确保工具之间正确地集成，例如能够将 UML 转换为 Java 代码，或进行相反的转换（如果希望这样）。这种思想当然可以用于其他技术平台，如 .NET。架构师还要确保拥有合适的基础架构，不仅用于支持开发工具，还用于支持正在考虑的系统的运行和测试。

10.1.8　架构师和业务分析

架构师可能想开展业务分析以理解组织中当前的问题，并确认有可能改进的领域。这可以通过实现一个自动化业务的某些方面的系统而提供这种改进。例如，这种系统可以解决如糟糕的效率和过多的人为错误等问题。

业务分析还可以确保客户、最终用户和开发人员对目标组织的结构理解一致。对组织中的角色、职责、各元素之间的关系的精确描述可以用于传达这种理解。这种理解经常以**目标运营模式**（Target Operating Model，TOM）或组织蓝图的形式出现。

为了构建组织需要的系统，可以在得到系统需求的同时进行业务分析。例如，某个系统可能需要和其他系统交互（以使业务的其他方面自动化）并允许组织内的某些角色访问（可能因此需要借助某些设备访问系统，如移动电话）。

另外值得注意的是，当业务的某一方面影响整个系统并有必要调整业务时，需要进行业务分析。一个很好的例子是，找出通用的业务任务和实体，最终在面向服务（SOA）架构中找出和业务有关的服务（service）。这类服务可以用于几个业务流程并可以在几个系统之间重用。

在第 7 章中我们对业务分析做了一些讨论，当时我们考虑了使用**业务流程模型**（Business Process Model）、**业务实体模型**（Business Entity Model）、**业务规则**（Business Rules）等工作产品。这些工作产品用于分析系统需求以支持组织的业务。在某些情况下，无法真正得到某些工作产品。理解系统所处的更广阔的环境是相当值得的，因此，对于系统的开发团队来说，唯一的选择就是进行一些业务分析。这就是为什么我们讨论这个规范。

所有的业务分析工作产品都会直接或间接地影响架构师的工作。这些工作产品影响系统需求，也因此直接影响架构师的工作。然而，架构师也要理解系统必须在其中运行的更大范围的环境，如果这些业务分析工作产品存在，那就参考它们。

10.2 架构师和外界影响

作为本书的焦点，IT 系统的架构并不是在孤立状态下开发的。如果系统是为了满足需求和解决利益相关者关心的问题，几种外界影响会引导对它的定义并必须纳入考虑。下面这 4 种外界影响总是需要在项目的上下文中加以考虑（包括它的架构）。这些影响是：

- 企业架构
- 设计权威
- 基础设施提供者
- 系统维护者

这些外界影响和他们与开发项目的关系，以及关联的系统如图 10.8 所示。我们会在后面的部分进行讨论。企业架构师和企业架构都包括在内。

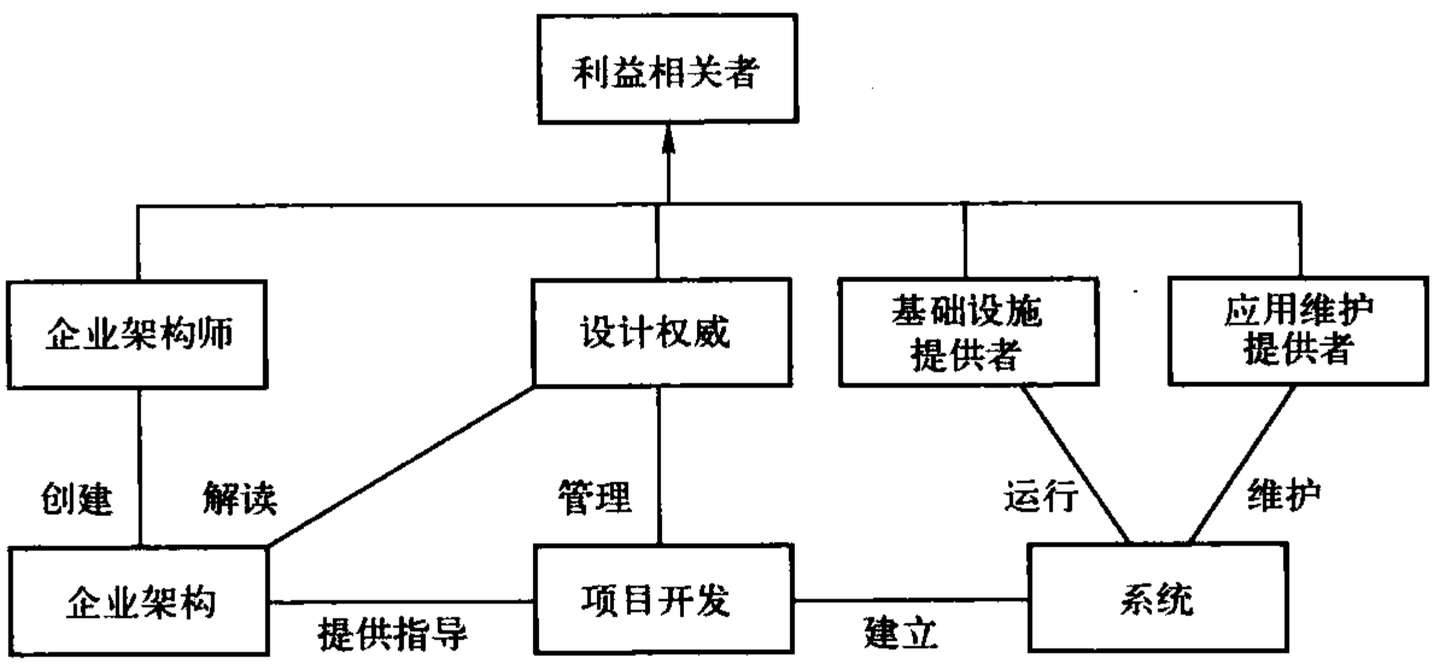


图 10.8 IT 系统架构的外界影响

图中显示的关系包括：

- 企业架构师、设计权威、基础设施提供者、应用维护提供者都是利益相关者。
- 企业架构师创建企业架构。
- 企业架构为项目开发提供指导。
- 设计权威管理项目开发。
- 开发项目建立系统。
- 设计权威解读企业架构规定的政策和规范。
- 当开发结束后，基础设施提供者运行部署后的系统。
- 系统部署后，系统维护提供者维护系统。

10.2.1　企业架构

根据 The Open Group Architecture Framework（TOGAF），开发企业架构的主要原因是：

> 通过提供一个 IT 策略的基本技术和流程结构来支持业务。这反过来使得 IT 成为一个成功现代企业策略的敏感资源。（TOGAF 2009）

尽管关于什么是企业架构还不存在业界广泛接受的定义，但大多数描述都赞同它涵盖 4 个领域：

- **业务架构**。这个领域包括策略、目标、政策、关键业务流程和业务组织（包括角色和组织结构）。
- **信息架构**。这个领域包括组织的关键信息资源，包括逻辑和物理数据模型、资源和元数据。
- **应用架构**。这个领域包括库存和描述应用软件和接口的逻辑、物理模型，以及它们如何支持组织的业务流程。
- **技术架构**。这个领域描述了支持系统开发的物理硬件和数据。它们通常包括中间件、操作系统和编程语言。

这不是一本关于企业架构的书，但是当一个组织已经定义了这种架构，并且所要构建的系统需要遵循企业架构涵盖的领域时，我们就会考虑它。我们在前几章介绍的好几个工作产品就归于企业架构的领域，并作为我们从第 7 ~ 9 章介绍的企业架构设计过程中各项任务的输入。

- **企业架构规范**。我们认为这些规范涵盖了我们刚才讨论过的一个或多个领域，并把它视为扮演设计权威的角色以确保项目开发团队遵循这些规范。
- **业务实体模型**。这个工作产品定义了企业中作为信息架构的一部分的业务实体。架构师使用这个工作产品的一部分来定义逻辑数据模型。
- **业务流程模型**。这个工作产品定义了组成业务架构的关键业务流程。
- **业务规则**。这个工作产品定义了管理业务运行的关键业务规则，也是业务架构中的一部分。
- **现有 IT 环境**。这个工作产品可能由企业架构师创建来作为应用和技术架构的一部分，它是对现有 IT 系统描述的分类。

10.2.2　设计权威

设计权威负责系统的总体技术完整性，并协调正在开发的系统和企业架构、策略、规范之间的关系以确保系统满足需求。设立单独的设计权威监管系统开发的关键原因之一是减少成本超支的风险、无法满足需求的风险或进度风险。一个设计权威可以以多种形式出现，这取决于系统的规模和复杂性：

- 对于小系统，设计权威可以就是**首席架构师**。系统足够小到一个人就可以理解，这个

人可以维护系统的总体技术完整性。

- 对于中等复杂程度的系统，设计权威可以由一个架构师专门担任并辅以一个或多个兼职架构专家向他或她汇报。
- 对于大型的复杂系统，设计权威可能由一个全职的架构团队组成，其中包括一个首席设计权威和各种角色的专家向团队汇报。

设计权威的职责多种多样，但是他们大多数承担以下职责：

- 通过定义和执行各种控制以确保所开发架构的一致性并能够满足需求。
- 确保符合各种内部的和外部的强制性标准和监管。
- 确保有一个有效的开发流程（和工具）并遵循这个流程。
- 向各种关心系统或对系统感兴趣的利益相关者负责。同时作为利益相关者和开发系统的团队之间沟通的桥梁。

设计权威有时候被称为技术设计权威，这并不恰当。然而，从作者们的观点来看，设计权威是一个业务和技术混合的角色。为了防止混淆，不应该使用术语**技术**。

本书描述的一些工作产品和设计权威有关，它们是：

- **企业架构规范**（Enterprise Architecture Principles），由企业架构师创建并需要遵循。
- **架构决策**（Architecture Decisions），设计权威参与决策。
- **架构评估**（Architecture Assessment），作为架构评估的结论（确保符合内部或外部标准的重要方法之一）。

10.2.3 基础设施提供者

当系统开发出来之后，需要在合适的基础设施上运行（也就是，在计算机平台、网络和其他硬件上）。提供基础设施的组织可以是构建系统的组织，也可以不是。在云计算时代，基础设施提供者经常有别于使用系统的组织。

为了知道运行系统需要的条件，基础设施提供者必须作为关键的利益相关者参与开发过程。基础设施提供者会补充**非功能性需求**工作产品（因为他们需要查看各种可能性）并参与创建**部署模型**，这是描述需要构建的基础设施的主要工作产品。基础设施提供者还参与架构复审并关心**架构评估**工作产品（是执行架构确认的结果）。

10.2.4 系统维护者

就像基础设施提供者可能来自于外部一样，在系统开发和部署之后，维护系统的人也可能来自外部，尤其是专业的维护人员可以维护多个系统以获得规模经济之后。然而，更有可能的情况是，系统由来自劳动力低廉的遥远地区的专业团队 24 小时远程维护。

系统维护者显然需要理解系统的细节，因此，他们应该尽早地提供对系统的需求。这些需求通常关注必须达到的质量，如可维护性、可移植性和可管理性，因为他们可能会指出可能存在的约束，或希望尽早知道维护系统需要的技能和资源。这样看来，系统维护者通常也是提供

非功能性需求工作产品输入信息的利益相关者之一。他们还会特别关心**功能模型**和**数据模型**，因为这些模型提供了对需要维护的系统和数据的高层描述。

10.3　复杂系统的架构设计

通过第7~9章的学习，我们已经看到架构师在为中等规模的项目（这种项目通常需要10~100人一起工作）设计一个相对简单的架构时可能会采用的方法。在这一部分，我们将会看到在设计复杂系统的架构时架构师所面临的一些挑战，例如涉及系统之系统的、通常会包括100名以上开发人员的那些系统。图10.9总结了这些复杂性和解决这些复杂性的最佳做法。

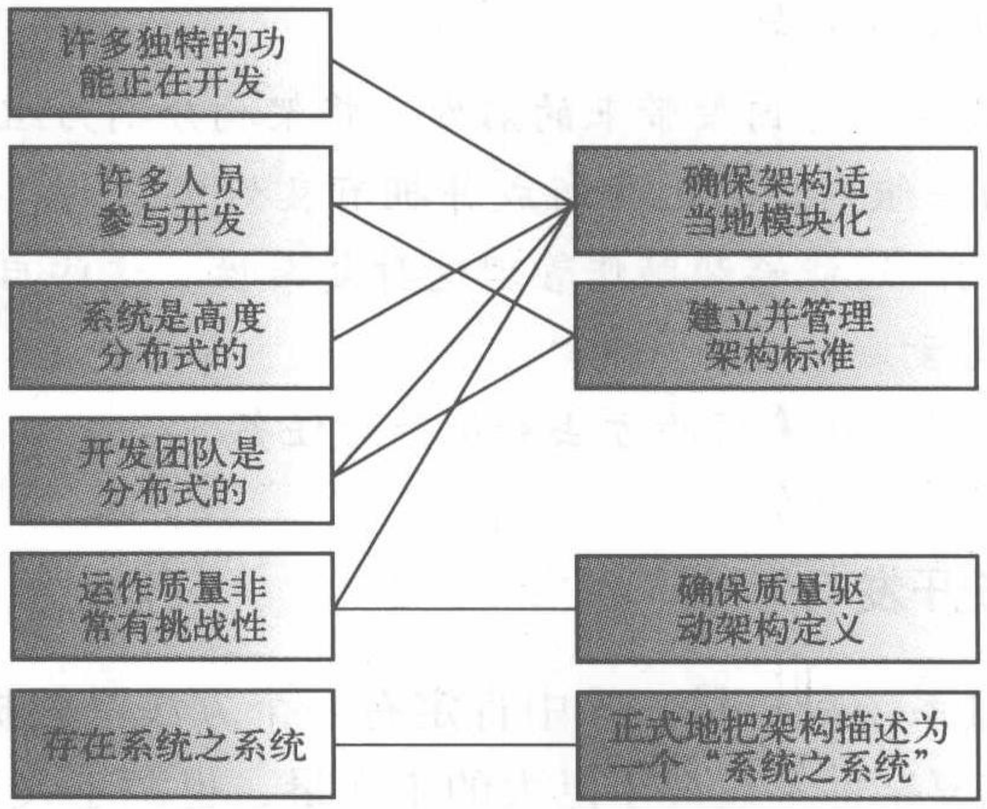


图10.9　复杂系统的挑战和相关的最佳方法

当然，这些特点并不一定是独立的。一个需要开发许多功能的项目也可能需要很多人来开发这些功能。然而，分离各种挑战能够使我们在接下来的部分分别描述它们。对于每项挑战，我们讨论以下内容：

- 为什么这个特点被视为一个复杂系统的属性
- 在开发这类系统的架构时需要额外考虑的内容
- 使架构师能够为这样的一个系统开发架构的架构任务的影响

10.3.1　许多独特的功能正在开发

规模大的系统并不一定复杂。例如，系统可能需要创建很多窗体，或描述许多交易。如果这些元素都遵循共同的模式，这种挑战并不复杂，只要一个人就能解决。然而，如果这些模式不容易看到，那么，这个系统由于其庞大的规模而注定复杂。这种情况会对开发工作的所有方面都产生影响，从需求到架构，到编码，到测试。

从架构和架构师执行的任务的角度来看，通过分离关注点来管理这种复杂性的做法是明智的。这个主题会在补充内容“最佳实践：确保架构适当地模块化”进行讨论。

最佳实践：确保架构适当地模块化

对架构进行正确的组块（chucking）或模块化会帮助您解决许多复杂系统带来的挑战。在前面的章节中，我们讨论了几类模块化技术：

- 同时从逻辑层面和物理层面考虑架构。特别是逻辑架构可以提供高视角的抽象，这能够帮助您分析复杂性的原因。
- 考虑分层的架构。不同的元素存在于不同的架构层中，恰当地安置各种架构元素将对您有所帮助。
- 考虑子系统。将架构分为多个子系统，每个子系统由内聚的元素组成，这能帮助您关注总体情况而不陷入细节。
- 确认组件时，考虑耦合与内聚带来的启发。将架构分割为组件（从最一般意义的角度），每个组件由一组内聚的元素组成并拥有良好定义的接口，从而能够让您在整个架构中重用它们。这能够帮助您管理这种复杂性。这些启发同样适用于高度分布式系统中的部署元素。

这些技术能够转变为架构师采用的方法和执行的任务。

10.3.2 许多人员参与开发

许多方法可以用来衡量系统的规模。其中肯定有一个方法是根据开发所需要资源的数量（人）来衡量系统复杂度。复杂性不仅源于巨大的工作量，也源于人员之间需要的沟通。如果沟通渠道不畅，重要的信息将无法传达给应该知道这些信息的人。另外，以正确的方式协调工作本身也很重要，所有的努力都应该用于确保每个人按照计划工作，在正确的时间按照正确的顺序提交各种项目工作产品。

当许多团队参与到项目中时，尽快地建立一个架构概览十分关键，它们能让我们分离各方面的关注点（例如每个团队负责一个子系统）并据此设计项目的结构。因为架构在一定程度上影响资源和工作的分配，例如，如果使用封装应用程序实现系统的主要部分，越早地确认这种情况越好，因为可能需要很多精力（因此还需要资源）用于定制这个封装应用程序。这项任务在特定时刻可能还需要某种特殊的技能。

这种情况也可能受益于适当地分离关注点，正如我们在前面的补充内容“最佳实践：确保架构适当地模块化”中所述。另外，为了确保架构的完整性，聪明的做法是设立架构标准并通过正确的管理来确保这些标准被采用，正如我们在补充内容“最佳实践：设立并管理架构标准”中所述。

最佳实践：设立并管理架构标准

为了确保架构在团队（可能会很大并且很分散）之间的完整性，设立所有团队成员在设计和开发解决方案时需要遵循的有关标准很重要。这种标准可以包括所采用的架构和设

计模式、建模标准和编程标准。设计权威可以执行这些标准，如果存在一个设计权威的话。

如果标准的推行导致工作产品按一致的方式创建（例如，所有组件必须暴露一个或多个接口），只要存在帮助开发的工具，自动遵循这些标准是有可能的。

为了将这种自动化看作是开发方法的基础，有一个特殊的业界标准。它是由对象管理组织（OMG）中的模型驱动架构（MDA）的先驱们提出的。MDA 定义了好几个概念，如图 10.10 所示。

MDA 定义了几种模型的分类：

- **独立于计算机的模型（CIM）**。这个模型定义了独立于解决方案的元素，不管解决方案是关于软件还是硬件或人。业务模型归于此类，包括**业务流程模型**和**业务实体模型**。
- **独立于平台的模型（PIM）**。这个模型定义了独立于技术的元素。**功能性需求**、**非功能性需求**、**逻辑功能模型**、**逻辑部署模型**归于此类。
- **特定平台模型（PSM）**。这一模型把技术纳入了考虑范围。**物理功能模型**、**物理部署模型**归于此类。
- **代码**。尽管通常不把它视为模型，组成系统的源代码也是某种抽象（在执行之前必须把它转换为机器码，即使采用了解释语言而非编译语言）。

转换是 MDA 强调的另一个概念。**转换**定义了从一个模型转换到另一个模型的规则，它还是一个强制一致性的好方法。这也会带来架构的完整性。图 10.11 列出了一个例子，其中一个 PIM 转换为一个采用 EJB 的 PSM（EJB 代表解决方案中粗粒度的组件）、一个表示 SQL 的 PSM（表述关系数据库中的持久元素）和一个描述 EJB 到 SQL 之间映射规则的 PSM（有时被称为对象—关系映射）。定义 EJB 的 PSM 可以转换为代码，就像定义 SQL 的 PSM 可以转换为对象—关系映射一样。每一个转换都有可能通过工具自动化实现。然而，即使是手工地进行这种转换，转换的概念仍然适用。

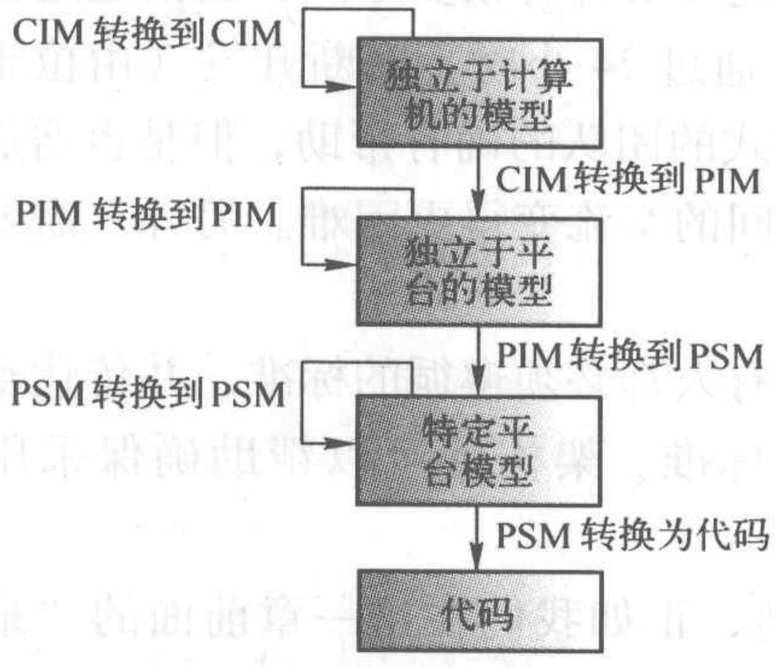


图 10.10　MDA 概念和术语

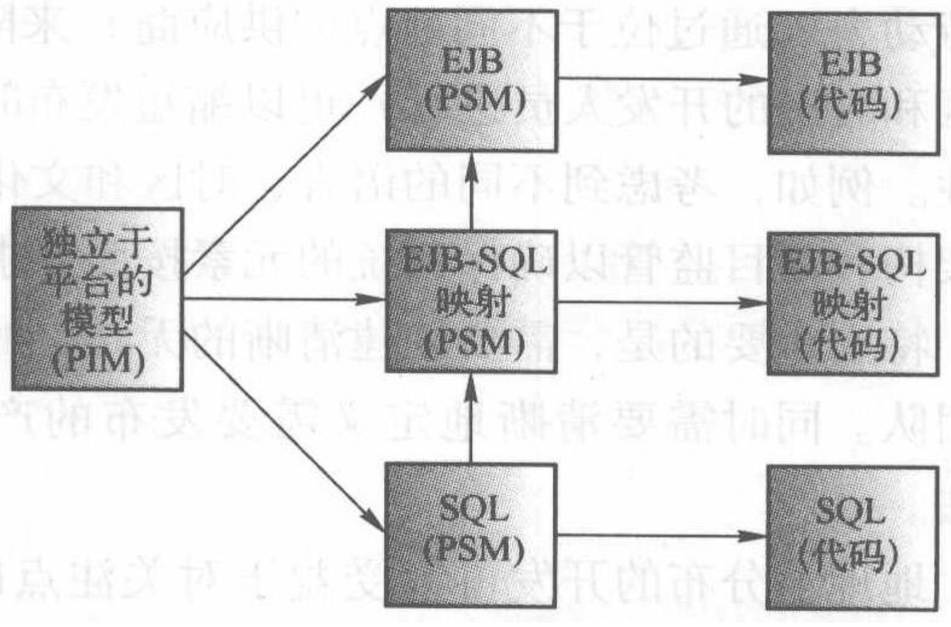


图 10.11　从 PIM 到 PSM 再到代码的转换

10.3.3 系统是高度分布式的

系统自身的部署可能跨越多个组织边界和地理边界。这类系统含有额外的复杂性，因为：

- 有额外的系统接口需要对付。这些接口用于支持系统的分布特性，系统的许多元素必须跨越分布式的环境相互通信。
- 可能需要考虑新的用户界面。一个地点可能会强制有一个地区性的界面。
- 可能需要考虑新的利益相关者。新的地域可能导致需要和为项目提供了输入信息的新的利益相关者沟通。
- 必须考虑新的操作内容。必须考虑提供合适的网络，质量变得更难以满足，如性能和可靠性。
- 时区的不同也会给依赖于时间的操作带来挑战。例如，如果选择批处理运行避开高峰时间，这个时间表会变得不清晰。还有，在每个工作日结束的时候可能需要执行某些操作（尤其在金融组织中），但是“每天的截止时间”的概念会变得模糊。
- 可能由于技术或监管的原因，业务数据需要必要的隔离。
- 跨越分布式环境的部署会涉及多个地点。每个地点都有细微的差别，这需要纳入考虑范围。

从架构来看，关注系统的部署方面可以帮助解决一些这样的挑战。例如，**部署模型**工作产品允许讨论各种部署元素，如地点、节点和它们之间的连接等。另外，例如，架构师以逻辑元素（如一个仓库）和物理元素（如奥兰多的仓库和坦帕的仓库）的形式对地点进行建模，这能够让我们在流程的早期考虑如何将功能元素安置在部署元素中。另外，对高度分布式的系统建模可以受益于对关注点的正确分离，正如我们在这一章前面的补充内容“最佳实践：确保架构恰当地模块化”所述。

10.3.4 开发团队是分布式的

开发复杂系统的团队经常分散在不同的物理地点。这样安排有很多原因，包括通过使用廉价劳动力（通过位于不同地点的供应商）来降低成本、通过 24 小时不间断开发（由位于不同地区和时区的开发人员进行）可以缩短发布时间。分布式的团队的确有帮助，但是也增加了复杂性。例如，考虑到不同的语言、时区和文化，人们之间的交流变得更困难。另外，需要恰当的架构和项目监管以确保系统的元素按照要求发布。

特别重要的是，需要创建清晰的无歧义的说明和所有人都必须遵循的标准，并传达给远方的团队。同时需要清晰地定义需要发布的产品和接受标准。架构师可以帮助确保采用这种方法。

地理上分布的开发可以受益于对关注点的正确分离，正如我们在这一章前面的“最佳实践：确保架构恰当地模块化”中所述。这种分离可以允许把系统进一步分割成子系统，这种方式可以有效地支持并行开发。另外，为了各个团队能够协调一致地工作并确保架构的完整性，

必须考虑采用相关的标准及合适的、一致的工具，正如我们在这一章前面的“最佳实践：设立并管理架构标准”中所述。

> 使用标准来表达架构中跨越地理边界的问题，这一技术可能延伸到组织边界，它可以变得和远程地理位置带来的问题一样严峻。这甚至可能仅仅涉及位于同一建筑的不同楼层的组织。如果架构构建不支持紧密沟通，小的地理距离也会变为大问题。(Coplien 2005)

当面对分布式的团队时，另一个焦点是协调和集成团队之间产品的方式。从协调的角度来看，这项任务很大程度上归结为项目管理，其中架构内容可以指导项目计划制定。架构还可以指导独立开发的项目元素的汇总、集成和测试的顺序。可能也关注验证任务和确认任务以确保分布式的团队交付正确的东西和以正确的方式交付。

10.3.5　运行质量非常有挑战性

一个系统需要展示出运行（运行期间）的质量，这极具挑战性。例如确保系统在99.99%的时间内正常运行的服务水平，在分布式系统上支持快速响应，或在敏感的时间范围转移大量数据，这些都具有挑战性。这种复杂的运行环境要求架构师平衡有时候会相互冲突的非功能性需求，请注意，这可能是在达到高性能的服务水平的同时还要按照给定的预算按时交付。

正如从第7~9章中所述，从一开始就密切地关注非功能性需求、这些需求的发展及解决这些需求的方法十分关键。对于具有困难的和有挑战性需求的系统，有更多的理由关注这些。例如，一个维持生命的机器如果不能满足稳定性需求，永远也看不到曙光。因此，这种需求能够代表正在开发的系统中最具挑战性的需求。结果，架构师会经常强调这种质量已经通过创建相映的架构概念证明得到证实。关于这些质量的重点会在后面的“最佳实践：确保质量驱动架构定义”部分进行讨论。

最佳实践：确保质量驱动架构定义

系统的架构不是完全关注系统提供的功能，运行的系统体现出的质量同样重要。架构师负责确保这些质量达到或超过系统需求。因此，确保记录、描述、考虑和在项目过程中核查这些质量，这对于架构师十分关键。

当一个系统必须满足一个具有特殊挑战性的需求或功能时，选择一种提供必要的关注点并能够使用适当的工作产品进行充实的视点是合适的。这些视点和相关工作产品的一些示例是：

- **可用性视点**。对于具有挑战性的可用性需求的系统来说，例如可能被数千甚至数百万用户使用的界面，可能存在一个完整的子项目来应对系统的这个问题。如果出现这种情况，那么我们可以选择一个可用性交叉视点并使用特定的可用性需求（Usability Requirements）和用户界面模型（User Interface Model）工作产品来充实这个视点。

- **性能视点**。在创建高性能的系统时，我们可以通过特定的性能模型（Performance Model）工作产品来充实我们在第 4 章“编写软件架构文档”中讨论的性能交叉视点。性能模型通过运用模拟、度量、建立基准线、经验法则来确认系统可能得到的性能，通过这些来证明系统是否能够满足所有的性能需求。
- **可靠性视点**。对于高可靠性的系统，我们可以使用特定的可靠性模型（Availability Model）来充实我们在第 4 章“编写软件架构文档”中描述的可靠性交叉观点。可靠性模型使用了多种建模技术（如可靠性图和故障树）来证实系统的可靠性。
- **安全性视点**。当特别关心安全性时，我们可以使用安全性需求（Security Requirements）、安全性架构（Security Architecture）和安全性威胁分析（Security Threat Analysis）工作产品来充实我们在第 4 章中描述的安全性交叉视点。

10.3.6 存在系统之系统

复杂系统的另一个特点是它们可能由子系统组成，这些子系统重要到它们自身也被视为一个系统。一位架构师负责的系统实际上可能也是一个更大系统的子系统。这两种情况都有可能，如图 10.12 所示。它显示了一个系统（系统 C）和其他相关系统的联系。

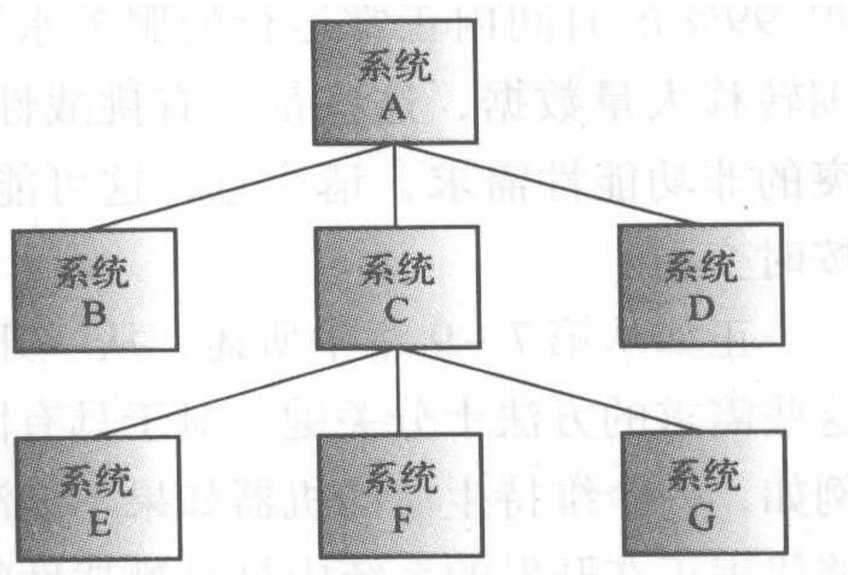


图 10.12 系统之系统

在图 10.12 中，系统 C 是系统 A 的子系统，它还拥有 3 个自己的子系统：E、F 和 G。这些系统可能都在它们自己的项目中开发并拥有自己的架构。当然，这些系统不是完全无关的——这正是我们在这一节要研究的内容。关于对系统之系统这个方法的确认将在补充内容“最佳实践：正式地把架构描述为系统之系统”中详细地讨论。

最佳实践：正式地把架构描述为系统之系统

把系统分解为许多子系统这个方法被广泛认可，并在《System of Interconnected Systems Architectural Pattern》（Jacobson 1997）中进行了介绍。这种模式可以用于描述系统和它的子系统之间的关系。在这种模式中，整个系统称为上级系统，而每个子系统称为下级系统。这个模式的一个重要特点是它是递归的，这意味着下级系统也可以拥有自己的子系统，相对这些子系统，它也是上级系统。互联系统之系统的架构模式可以用于许多方案，例如下面这些：

- **面向服务的架构（SOA）**。在这种情况下，SOA 自身能够看作是一个系统（通常是企业范围的或组织范围的），每一个服务代表一个子系统。
- **企业架构**。正如我们在第 2 章“架构、架构师、架构设计”中所述，企业架构考虑硬件、软件和人等各种元素以及与业务对象有关的信息，例如业务的敏捷性和组织

效率。从这个模式来看，企业代表整个系统，企业的元素代表子系统。

- **系统工程**。类似地，系统工程涉及硬件、软件、人和信息。系统作为整体可以分解为子系统。这些子系统也可能包含这些不同类型的元素。
- **战略性重用**。根据定义，可重用资产不会孤立存在（因为它们要在好几个相互联系中重用）。因此，任何战略性重用动机的一个基本前提是定义每一个资源提供的服务和这个资源需要的服务。这些资源和它们之间的关系能够以系统之系统的方式进行描述。容纳资源的资源代表系统，可重用资源代表子系统。这种思想和一个软件生产线（代表一组应用于特定工业部门的通过可重用资源创建的系统）的创建尤其相关。和软件生产线相关的内容在《Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach》（Bosch 2000）和《Software Product Lines: Practices and Patterns》（Clements 2001）中有详细的讨论。
- **应用集成**。应用集成关注于开发一个包含好几个现有系统的解决方案。这些工作能够由下向上推广到更大的范围，因为解决方案的元素已经存在，但是，仍然需要一些由上向下的工作来理解这些现有系统所适应的上下文。通常需要类似于封装到现有软件应用接口的技术，还要对可以用于和这些系统交互的中间件技术有很好的理解。关于互联系统之系统的架构模式，集成的企业应用看作是系统整体，而每一集成应用看作是一个子系统。
- **打包应用开发**。打包应用可以看作是允许您构建支持一个业务特定方面（如客户关系管理或人力资源）的一组相关应用的可定制框架。这种框架可以分为两个级别。第一种，这类框架通常用于实现一个大系统的一部分。在这种上下文中，打包应用（或它的一部分）代表一个子系统。第二种，这种框架通常大且复杂。把它们自身想象为系统可以帮助您理解如何应用打包应用程序：哪些部分可以直接应用，哪些部分将在修改和配置后使用，哪些部分根本不使用。实际上，这些应用经常分割为模块，每个模块拥有本身就可以看作是子系统的特定的功能。

在描述系统之系统时，需要接纳各种各样的架构规范，其中比较形式化的一种可以在《Model-Driven Systems Development approach from IBM Rational》（MDSD 2009）中找到，这也是以前的系统工程 RUP（RUP SE），属于 *Rational Unified Process*（RUP 2008）的一部分。这是 Murray Cantor 博士和其他人的成果。MDSD 特别支持系统工程方案（但是也能够用于其他方案），其中系统定义如下：

> **系统**是一组提供由企业用于实现业务目标或任务的服务的资源。系统组件通常由硬件、软件、数据和工人组成（Cantor 2003）。

MDSD 方法的一个优点是，它是一组用来把系统需求分配给子系统的技术（包括功能性需求和非功能性需求），其中的子系统本身也可以包括软件、硬件、人和数据。联合实现（joint realization）是其中的关键技术之一，也是 MDSD 的基石之一。这项技术描述了如何跨越组成

系统的不同类型的元素（从软件、硬件、人和数据等方面）来联合实现系统的功能性需求和非功能性需求。MDSD 试图正面解决当单独考虑这些元素时会出现的广为人知的问题。

应用 MDSD 方法的结果是一组松散耦合并高度内聚的子系统，每一个子系统都拥有一组能够让我们把其自身看作是一个系统的需求（需要满足的功能性需求、需要体现的质量、需要接受的约束）。

10.4 总结

这一章通过介绍本书中还没有讨论的主题补充了前面的章节，这些主题超出了对架构师在一个典型软件开发项目中的角色的基本理解。

我们特别讨论了架构师在几个我们在本书的前面部分还未曾讨论过的软件开发规范中的作用，还讨论了架构师在面临复杂系统时所做的额外思考。然而，对于解决这些复杂的挑战，我们介绍的仅是皮毛。这一领域已经受到了业界应有的关注。

最后：作者们的提示

正如我们在本书开始时所说，我们的目的是介绍一些您可以应用到整个或部分架构设计流程中的最佳方法。我们设法构建一个每位架构师都应该掌握的关键知识的基础，随着本书的展开，这些内容也逐渐清晰起来。最后，我们讨论了一些更复杂的话题，这些话题本身都值得写一本书来进行讨论。

不管您如何使用本书的内容，不管您是一位有抱负的架构师还是一位高级的使用者，我们都希望您能成功地应用这些方法。最后请记住，即使是最佳方法也能进一步改进！

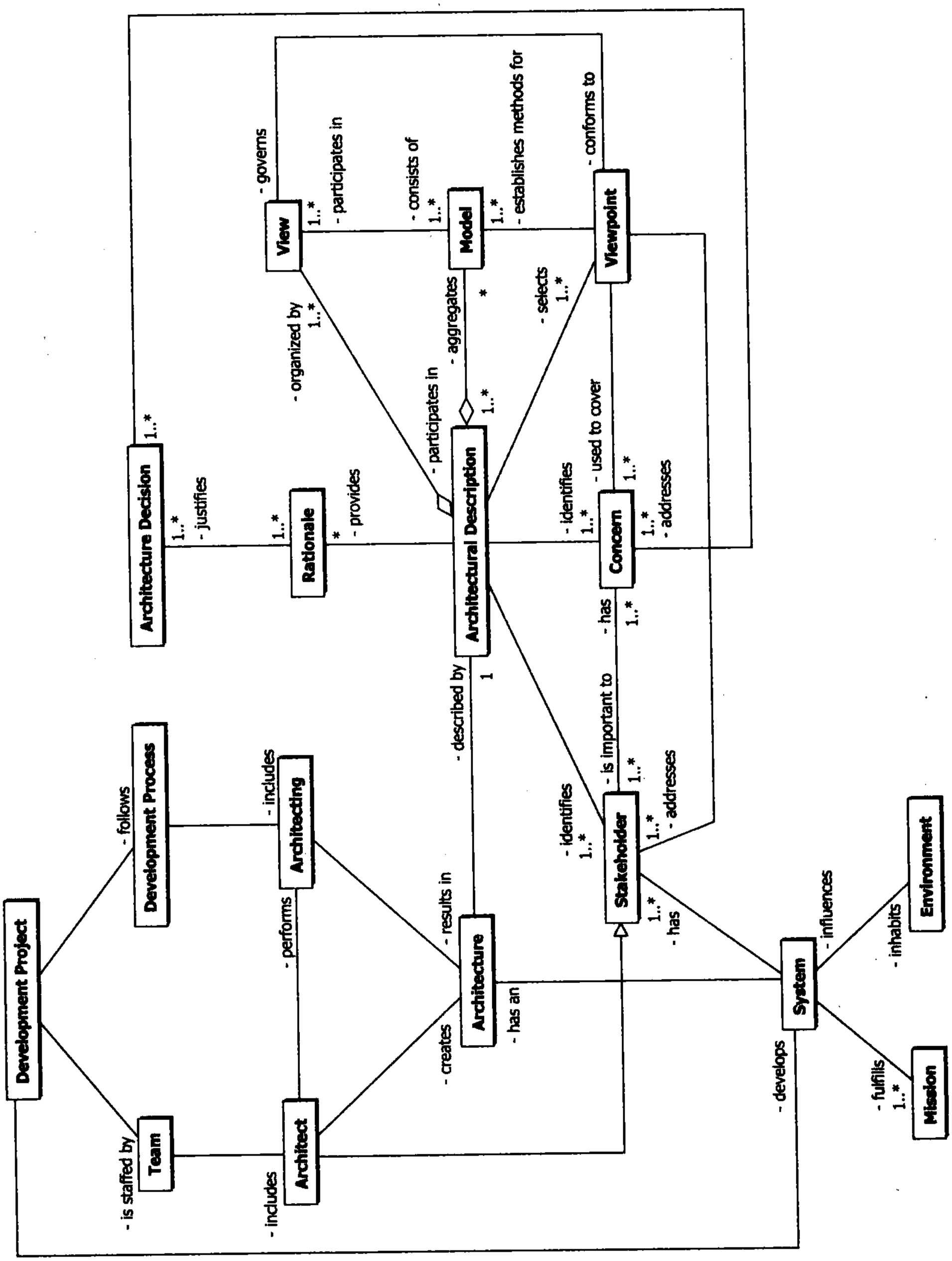
附录 A

软件架构元模型

这个附录包含了本书中使用的全部概念的元模型的定义。

图 A.1 中显示的元模型关系是：

- 一个系统拥有一个架构。
- 一个系统完成一个或多个任务。
- 一个系统拥有一个或多个利益相关者。
- 一个系统存在于一个环境中。
- 一个环境影响一个系统。
- 一个架构由一个架构描述来描述。
- 一个架构描述确定一个或多个利益相关者。
- 一个架构描述确定一个或多个关注点。
- 一个架构描述提供一个基本原理。
- 一个利益相关者拥有一个或多个关注点。
- 关注点对于一个或多个利益相关者很重要。
- 一个开发项目由一个团队来担任。
- 一个开发项目遵循一个开发流程。
- 一个开发项目开发一个系统。
- 开发流程包括架构设计。
- 团队包括一个架构师。
- 架构师进行架构设计。
- 架构师创建架构。
- 架构师是一类利益相关者。
- 架构设计产生一个架构。
- 基本原理验证一个或多个架构决策。
- 一个架构决策处理一个或多个关注点。
- 架构描述是按一个或多个视图来组织的。
- 一个视图由一个或多个模型组成。
- 一个模型参与一个或多个视图。


Development Project
- follows
Development Process
- includes
Architecting
- performs
Architect
- includes
- is staffed by
Team
- results in
Architecture
- creates
- has an
Architectural Description
- described by
1
- participates in
- organized by
1..*
View
- governs
- participates in
- consists of
Model
- establishes methods for
- conforms to
Viewpoint
- selects
- identifies
Stakeholder
- is important to
- has
Concern
- used to cover
- addresses
- identifies
- has
- addresses
- justifies
Architecture Decision
- provides
Rationale
- influences
- inhabits
Environment
- develops
System
- fulfills
Mission

图A.1 软件架构元模型

- 一个模型参与一个或多个架构描述。
- 一个视图遵循一个视点。
- 一个架构描述选择一个或多个视点。
- 一个视点为一个或多个模型建立方法。
- 一个视点用于涵盖一个或多个关注点。

元模型术语的定义

下面讲述这个元模型中使用的术语的定义。

architect（架构师） 负责系统架构的人、团队或组织。（IEEE 1471 2000）这个术语和软件架构师是同义词。

architecting（架构设计） 对一个架构进行定义、编写文档、维护、改进和确保适当实现的活动。（IEEE 1471 2000）这个术语和软件架构设计是同义词。

architectural description（架构描述） 编写一个架构的文档集。（IEEE 1471 2000）

architecture（架构） 体现在系统的组件中、它们彼此的关系中、与系统环境的关系中及指导系统设计和进化的原则中的一个系统的基本组织。（IEEE 1471 2000）

architecture decision（架构决策） 关注一个软件系统整体或系统的一个或多个核心组件的有意的设计决策。这些决策确定系统的非功能性特性和质量因素。（Zimmermann 2008）

concern（关注点） 与系统的开发、操作或对于一个或多个利益相关者来说关键或重要的其他方面相关的注意点。关注点包括像性能、可靠性、安全性、分布性及可发展性等这样的系统考虑。（IEEE 1471 2000）

development process（开发流程） 在系统开发过程中执行的、有特定目的的一组部分顺序化的步骤，例如构建模型或实现模型。（UML 2.0 2003）

development project（开发项目） 项目是一个承诺创建一个独特产品或服务的临时性意图。临时性的意思是每个项目都有一个明确的起点和一个明确的终点。独特的意思是这个产品或服务和所有类似的产品或服务存在差异。项目通常是进行组织业务战略的关键组成部分。（RUP 2008）

environment（环境） 环境（或上下文）决定对系统的开发性、操作性、政策性或其他方面产生影响的情况和设置。（IEEE 1471 2000）

mission（任务） 一个或多个利益相关者通过使用系统来达到的一组目标。（IEEE 1471 2000）

model（模型） 模型提供一个环境的特定说明或内容。例如，一个结构性视图可能由一组系统结构模型组成。这些模型的元素可能包括可确认的系统组件和它们的界面，以及在这些组件之间的联系。（IEEE 1471 2000）

rationale（基本原理） 一组行动或一个信念的一组理由或逻辑基础。

stakeholder（利益相关者） 对系统关注或感兴趣的一个人、团队或组织（或其中的某

些层级）。

system（系统）（1）软件系统是组织在一起来实现一个或一组功能的组件集合。系统这个术语包括单独的应用程序、传统意义上的系统、子系统、系统之系统、产品线、产品族、整个企业及其他利益组合。一个系统的存在是为了在它的环境中完成一个或多个任务。（IEEE 1471 2000）

（2）由一个或多个流程、硬件、软件、设施和人员组成的提供一个能力来满足一个特定需要或目标的一个完整的复合物。（IEEE 12207 1995）

team（团队） 致力于一个共同目标、业绩指标和采用方法相互负责的一部分技能互补的人。（Katzenbach 1993）

view（视图） 把整个系统看作是一组相关关注点的一种表示法。（IEEE 1471 2000）

viewpoint（视点） 构建和使用一个视图的一个传统规范。通过确定视图目标和拥护者及确定创建和分析该视图所用的技术来开发不同视图的模式或模板。（IEEE 1471 2000）

附录 B

视 点 目 录

这个附录概括了我们在本书的第 4 章中使用的视点。在那章中，我们讨论了基础视点和交叉视点的概念。

架构视点代表构建和使用一个视图的传统规范。用来通过确定视图目标和拥护者及确定创建和分析该视图所用的技术来开发不同视图的一个模式或模板。(IEEE 1471 2000)

图 B.1 中列出了各种各样的视点，这个图是从第 4 章复制过来的。

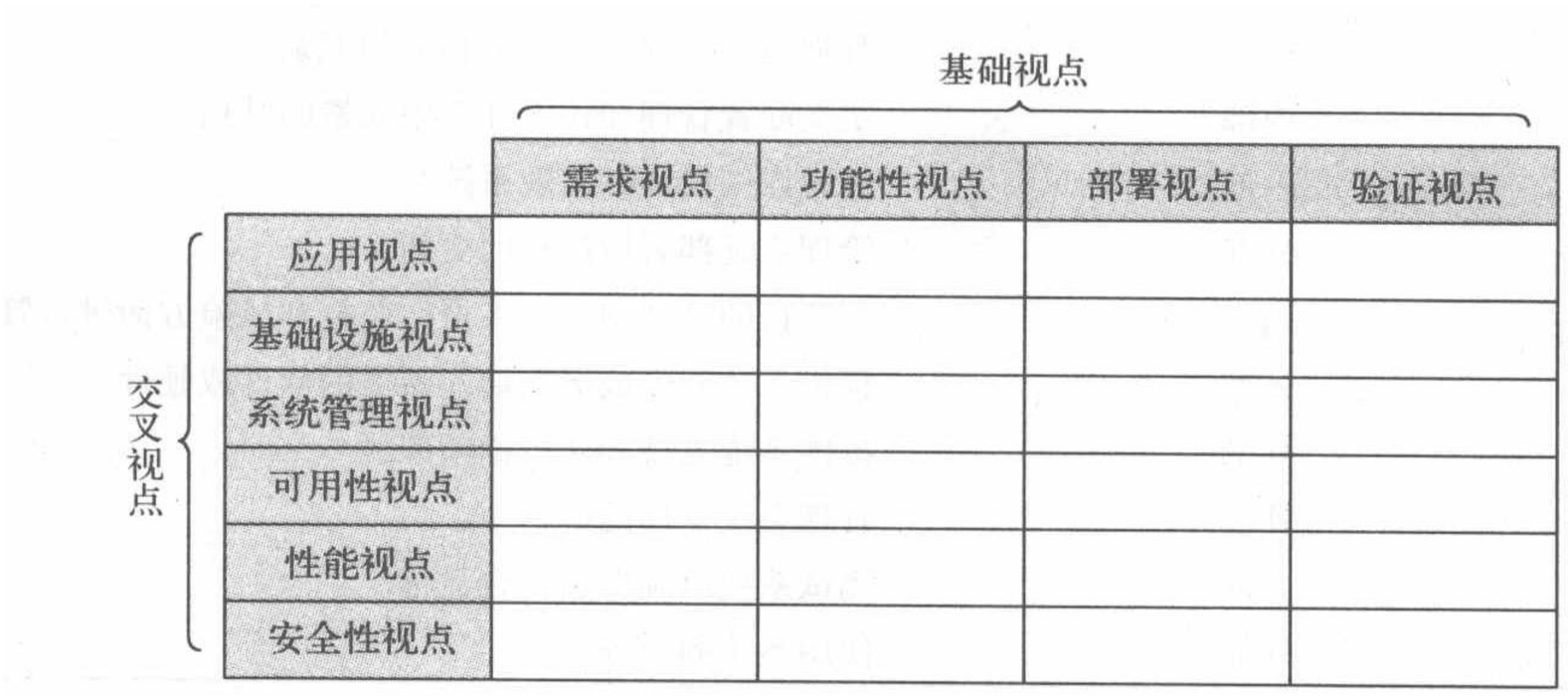


图 B.1　架构描述框架

第 4 章用一个模板来描述每个视点，包括下列项目：

- **描述**。该视点的概要描述。
- **利益相关者关注点**。该视点处理的利益相关者的关注点。
- **利益相关者**。该视点处理的利益相关者。
- **工作产品**。由该视点定义的视图所引用的工作产品。
- **举例说明**。一个或多个例子。
- **检查列表**。当检验和确认架构或与利益相关者一起复审架构时可以考虑的一个或多个问题。

这个模板中明显忽略了在应用这个视点时可能会使用到的技术的详细讨论。虽然在先前相关的章节中（第 7 ~9 章）已经讨论了一些技术，但是，在这个附录中是有意忽略某些特定的技术。关于需求定义、性能建模、安全性管理等方面的详细信息，有许多优秀的书可以参考。

在考虑这个目录时，您应该记住，在本书中使用的架构描述标准可以根据需要进行精炼和扩展。例如，如果您认为混合信息、并行性和其他视点等内容对于开发中的系统很有意义，那就可以查看它们。

利益相关者概述

这个目录中的每个视点都列出了由其处理的利益相关者。表 B.1 概述了这个目录中考虑的利益相关者。在确定特定情况下架构的利益相关者时，您可以从这个概述着手。这个表还指明了每个利益相关者的范围，内部利益相关者是在当前开发团队内的人，而外部利益相关者并不是。虽然架构师没有包含在这张表中，但是，我们假定架构师是所有视点中的一位利益相关者。

表 B.1 利益相关者概述

角 色	范 围	主 要 职 责
应用程序拥有者	外部	委托并为系统付费
业务管理员	外部	管理运行系统中与业务相关的元素
配置经理	内部	定义配置管理知识库和其中元素的结构
开发人员	内部	进行详细设计并实现系统
维护人员	外部	管理系统部署后进行的更新
项目经理	内部	在项目的计划编制、人员、监控和风险方面进行管理
供应商	外部	提供系统开发或运行期间所需的软件或硬件
支持人员	外部	提供系统支持和履行诊断职能
系统管理员	外部	管理系统操作的运行
测试人员	内部	测试系统以确保它符合目标
使用人员	外部	使用系统的业务功能

基础视点

这一部分包含了对基础视点的描述。

需求视点

描述 这个视点关注那些形成架构的系统需求，包括功能性需求、质量和约束。

利益相关者关注点 架构上重要的功能性需求、必需的质量、解决方案约束、需求优先级和外部参与者。

利益相关者 所有的利益相关者。

工作产品 业务实体模型、业务流程模型、业务规则、变更请求、企业级架构规则、现有 IT 环境、功能性需求术语表、非功能性需求、排定优先级的需求列表、利益相关者请求、系统上下文和愿景。

检查列表 需求是否已经确定并明确到用来定义架构解决方案的详细程度？是否像功能性需求一样，已经考虑了非功能性需求（质量和约束）？

功能视点

描述 这个视点关注那些支持系统功能性（逻辑上的）的架构元素，包括构建系统的结构元素、它们之间的关系和它们的行为。

利益相关者关注点 系统的功能、与最终用户和外部系统的接口、主要的结构性组件、组件之间的关系和行为、数据结构、数据流、数据存取和事务管理。

利益相关者 所有的利益相关者。

工作产品 架构决策、架构概览、数据模型和功能性模型。

检查列表 相应的功能性元素是否支持规定的功能性和非功能性（质量和约束）需求？特定的约束包括与这些功能性元素实现相关的成本和计划。

功能性元素是否可追溯到需求？

所有的组件都展现出适当的内聚性、耦合性和粒度吗？

任何组件的接口都包含一组内聚的有验证签名和前置条件及后置条件的操作吗？

所有的子系统都代表一组内聚的能力吗？

持久的功能元素定义了相应的数据元素了吗？

部署视点

描述 这个视点关注那些支持系统发布的架构元素，包括像位置、节点、设备和它们之间的连接这样的元素。

利益相关者关注点 系统分布、硬件节点（和设备）规范和网络规范。

利益相关者 配制经理、开发人员、维护人员、项目经理、供应商、支持人员、系统管理员和测试人员。

工作产品 架构决策、架构概览和部署模型。

检查列表 相应的部署元素支持规定的功能性和非功能性需求吗？特定的约束包括从现有的环境中迁移、成本、实现进度（包括获取必需硬件的时间）和物理约束（例如在一个数据中心中的可用空间）。

部署元素可以追溯到需求吗？

功能视图中所有组件都已经部署到部署视图中的节点上了吗？

节点和地点之间是否存在连接来支持功能性元素之间的交互？

物理部署元素在节点和设备的数量和它们详细的规范方面是否已经定义得足够详细？

第三方软件的所有需求（如操作系统）及需要软件许可协议的运行期元素都已经确定了吗？

验证视点

描述 这个视点关注于评估系统是否会提供必需的功能、达到要求的质量及适应定义的约束。

利益相关者关注点 架构的恰当性、可替代解决方案的考虑及技术风险和问题的解决办法。

利益相关者 应用程序拥有者、项目经理和测试人员。

工作产品 架构评估、架构概念证明、RAID 日志和复审记录。

检查列表 架构中已经采用了架构决策了吗?

确认（和包括）了适当的可重用资源了吗?

已经选择的所有的第三方元素（如应用程序包和基础设施产品）符合公司的政策吗（例如，这些应用程序包或软件产品及供应商包含在首选的供应商/产品清单中吗)?

架构和任何强制性的政策、标准和方针一致吗?

架构在可以认为等同于生产环境的开发或测试环境中确认过吗?

在可能的情况下，规定的需求（尤其是像性能这样的质量需求）在一个可执行环境中确认过吗?

考虑过可替代的解决方案吗?

有没有分析过技术性风险并进行适当的缓解?

交叉视点

这一部分描述了所有的交叉视点。

应用视点

描述 这个视点关注那些提供解决方案中特定应用行为（就是系统必须提供以满足它的业务需求的行为）的架构元素。

利益相关者关注点 应用的特定需求和它们的实现。

利益相关者 业务管理员、开发人员、维护人员、供应商、系统管理员和测试人员。

工作产品 这个视点是交叉的，可能引用所有的工作产品。

检查列表 这个视点是交叉的，所有的基础视点中的检查列表都适用于这个视点的上下文中。

基础结构视点

描述 这个视点关注那些提供解决方案中与应用无关的行为（就是系统必须提供以支持系统业务为目的的行为）的架构元素。

利益相关者关注点 与应用无关的需求和它们的实现。

利益相关者 开发人员、维护人员、供应商、系统管理员和测试人员。

工作产品 这个视点是交叉的，可能引用所有的工作产品。

检查列表 这个视点是交叉的，所有的基础视点中的检查列表都适用于这个视点的上下文中。

系统管理视点

描述 这个视点关注那些当系统部署到生产环境时有助于其操作运行的架构元素。

利益相关者关注点 系统启动和关闭、操作系统的监视和控制、数据备份、数据归档和数据恢复。

利益相关者 开发人员、维护人员、供应商、系统管理员和测试人员。

工作产品 这个视点是交叉的，可能引用所有的工作产品。

检查列表 这个视点是交叉的，所有的基础视点中的检查列表都适用于这个视点的上下文中。

关于系统启动和关闭、操作系统的监视和控制、数据备份、数据归档和数据恢复的需求已经处理了吗？

可用性视点

描述 这个视点关注那些使系统能够达到指定可用性需求的架构元素。这个视点还关注那些使系统从故障（使系统全部或部分不可用）中恢复的架构元素。

利益相关者关注点 系统启动和关闭、服务质量和系统恢复。

利益相关者 开发人员、维护人员、供应商、系统管理员和测试人员。

工作产品 这个视点是交叉的，可能引用所有的工作产品。

检查列表 这个视点是交叉的，所有的基础视点中的检查列表都适用于这个视点的上下文中。

关于系统启动和关闭、服务质量和系统恢复的需求已经处理了吗？

在相关的地方有没有考虑不同的服务质量？

性能视点

描述 这个视点关注那些使系统能够达到指定性能需求（如响应时间和生产能力）的架构元素。

利益相关者关注点 响应时间、生产能力和可量测性。

利益相关者 开发人员、维护人员、供应商、系统管理员和测试人员。

工作产品 这个视点是交叉的，可能引用所有的工作产品。

检查列表 这个视点是交叉的，所有的基础视点中的检查列表都适用于这个视点的上下文中。

关于响应时间、生产能力和可量测性的需求已经确定并处理了吗？

安全视点

描述 这个视点关注那些使系统能够满足指定安全需求的架构元素，例如授权访问系统的功能和资源。这个视点还关注那些使系统能够在系统的安全机制内发现故障并从故障中恢复的架构元素。

利益相关者关注点 安全性威胁、安全策略和威胁侦测。

利益相关者 开发人员、维护人员、供应商、系统管理员和测试人员。

工作产品 这个视点是交叉的，可能引用所有的工作产品。

检查列表 这个视点是交叉的，所有的基础视点中的检查列表都适用于这个视点的上下文中。

关于安全性威胁、安全策略和威胁侦测的需求已经处理了吗？

保护的资源是按这样确定的吗？

安全策略已经确定并声明了吗？

违背安全性的行为及如何恢复的方法定义了吗？

视图对应

在先前的视点描述中没有讨论由某一视点定义的视图和其他视图之间的关系。例如，一个组件在节点上的位置既是功能性视图的关注点，也是部署视图的关注点。您可能要把功能性视图中确定的所有组件都部署到部署视图中的节点上。

我们提供表 B.2 来把重复性最小化，而没有在先前的每一部分中都描述视图的对应关系。这个表指明了架构描述框架中所有视图的对应关系，其目的是使您通过查看您正在使用的每个视图的所在列来了解必须存在什么对应关系（及完整性和一致性）。

表 B.2 视图对应

	需 求	功 能	部 署	确 认
需求		功能性元素如何实现需求（例如，订单处理组件）？	部署元素如何实现需求（例如，允许用户下订单的一个节点）？	架构对于需求的满足程度如何（例如，对下订单的解决方案的评估）？
功能	功能性元素如何实现需求（例如，订单处理组件）？		功能性元素放置在哪些部署元素上（例如，订单处理组件所放置的节点）？	功能性元素对于需求的满足程度如何（例如，对订单处理组件的评估）？
部署	部署元素如何实现需求（例如，允许用户下订单的一个节点）？	功能性元素放置在哪些部署元素上（例如，订单处理组件所放置的节点）？		部署元素对于需求的满足程度如何（例如，对下订单所需节点的评估）？
确认	架构对于需求的满足程度如何（例如，对下订单的解决方案的评估）？	功能性元素对于需求的满足程度如何（例如，对订单处理组件的评估）？	部署元素对于需求的满足程度如何（例如，对下订单所需节点的评估）？	

（续）

	需　求	功　能	部　署	确　认
应用	什么是应用特定的需求（例如，下订单的用例）？	什么是应用特定的功能性元素（例如，订单处理组件）？	什么是应用特定的部署元素（例如，允许用户下订单的节点）？	应用特定需求的满足程度如何（例如，下订单的解决方案的评估）？
基础设施	什么是应用无关的需求（例如，使用某一特定数据库的约束）？	什么是应用无关的功能性元素（例如，一个持久机制）？	什么是应用无关的部署元素（例如，一个打印机）？	应用无关需求的满足程度如何（例如，评估使用特定的数据库的约束是否已经满足）？
系统管理	什么是系统管理的需求（例如，系统在失败的情况下应该报告一个异常）？	什么是系统管理相关的功能性元素（例如，支持错误通知的组件）？	什么是系统管理相关的部署元素（例如，支持系统管理员的节点）？	系统管理相关的需求的满足程度如何（例如，系统失败后报告异常的解决方案的评估）？
可用性	什么是可用性需求（例如，系统将达到99.99%可用）？	什么是影响可用性的功能性元素（例如，支持离线工作的组件）？	什么是影响可用性的部署元素（例如，支持系统故障恢复的节点）？	可用性需求的满足程度如何（例如，系统99.99%可用的解决方案的评估）？
性能	什么是性能相关的需求（例如，一次支付必须在5秒内处理）？	什么是影响性能的功能性元素（例如，数据压缩涉及的组件）？	什么是影响性能的部署元素（例如，局域网和广域网的连通性）？	性能相关需求的满足程度如何（例如，在5秒内处理一次支付的能力的评估）？
安全	什么是安全性相关的需求（例如，所有的用户都必须是认证的）？	什么是安全性相关的功能性元素（例如，提供用户认证的组件）？	什么是安全性相关的部署元素（例如，阻止未认证用户访问的防火墙）？	安全性需求的满足程度如何（例如，用户认证内容的评估）？

附录 C

方 法 概 述

这个附录概述了我们在本书中参考的方法。第 3 章“方法基本原理”中提供了与方法相关的基本概念的概览。

角色

角色定义了软件开发组织内作为团队一起工作的一组人或单个人的职责。角色负责一个或多个工作产品并执行一组任务。

项目团队内的角色（如本书所述）如图 C.1 所示。

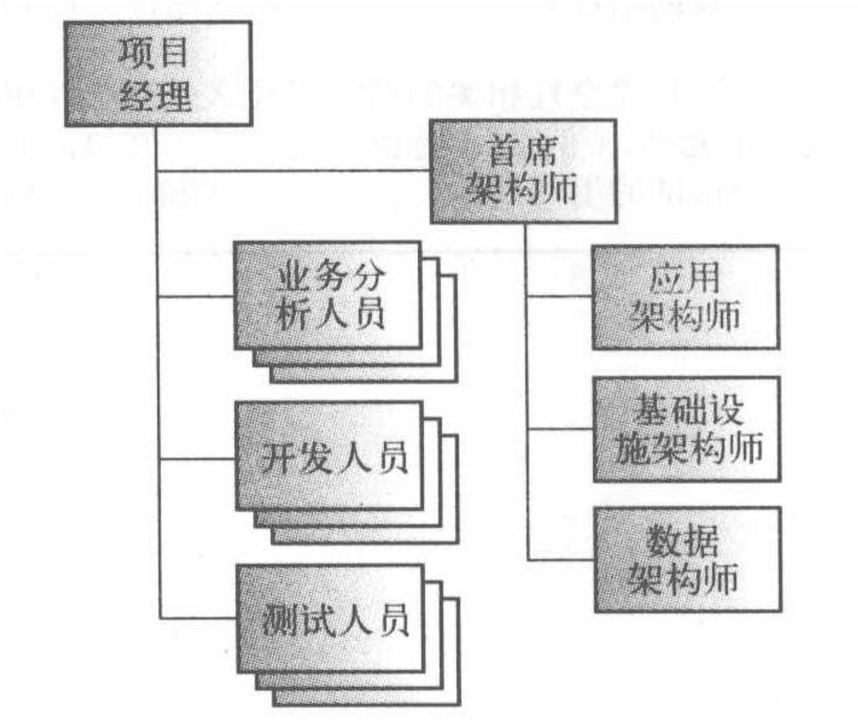


图 C.1　项目团队

表 C.1 提供了项目团队中角色的概述。

表 C.1　角色概述

角色名称	描　　述	拥有的工作产品
应用架构师	应用架构师关注那些使业务流程自动化和满足业务需要的元素。这个角色主要关注业务需要的功能，但是，他也关心应用相关的元素如何满足系统的非功能性需求（质量和约束）	功能性模型
业务分析人员	业务分析人员负责确定和编写需求文档及从业务的视角分析这些需求。担任这个角色的人员定义当前和将来的运作场景（流程、模型、用例、计划和解决方案）并与利益相关者和架构师一起工作来确保从业务需求到系统需求进行适当的转移	业务实体模型、业务流程模型、业务规则、企业架构规则、现有 IT 环境、功能性需求、术语表、非功能性需求、排定优先级的需求清单、利益相关者请求、系统上下文、愿景

（续）

角色名称	描　述	拥有的工作产品
数据架构师	数据架构师关注系统的数据元素，尤其是那些用适当的机制（如数据库、文件系统、内容管理系统或其他存储机制）保持持久的数据。这个角色定义适当的与数据相关的属性，如结构、来源、位置、完整性、可用性、性能和使用年限	数据模型
开发人员	开发人员负责进行详细的设计任务、定义实现的组织、实现详细的设计元素、对实现进行单元测试、把他的工作和其他开发人员的工作集成起来并最终形成可执行的系统	开发人员不拥有本书中讲述的任何工作产品
基础设施架构师	基础设施架构师集中关注那些不依赖业务功能的系统元素，如持久机制、硬件和中间件。这样的元素支持应用相关元素的执行。这个角色关注那些对系统质量有明显影响的元素，因此，他也会处理一些延伸的非功能性需求	部署模型
首席架构师	首席架构师全面负责定义系统架构的主要技术决策。这个角色也负责为这些决策提供理论基础；平衡各种利益相关者的关注点；管理技术风险和问题；还确保决策被有效地沟通、验证和执行	架构评估、架构决策、架构概览、架构概念证明、软件架构文档
项目经理	项目经理通过应用一些项目管理原则、工具和技术来负责领导将交付解决方案的项目团队。项目经理整体负责管理范围、成本、进度和结构性交付物，也负责管理问题、风险和变更	变更请求 RAID 日志 复审记录
测试人员	测试人员负责发现软件中的缺陷和编写测试文档、对察觉的软件质量问题提出警告，还根据设计和需求来验证系统功能。作为对开发人员进行的单元测试的补充，测试人员关注开发人员提供的集成单元和系统整体	测试人员不拥有本书中讲述的任何工作产品

工作产品

工作产品是在流程执行过程中生成和/或使用的一些信息或物理实体。工作产品的例子包括模型、计划、代码、可执行代码、文档、数据库等。本书中用到的工作产品如表 C.2 所示。

表 C.2　工作产品概述

工作产品名称	描　述	拥有的角色
架构评估	记录了在项目生命周期的不同阶段对解决方案架构的评估的结果。这些评估确认和架构或架构需求相关的问题和风险并阐述解决这些问题和风险的行动和缓解策略	首席架构师
架构决策	记录了在塑造整体架构时做出的重要决定，还描述了进行这些决策的理由、考虑的各种选项和这个决策的影响	首席架构师
架构概览	提供了架构关键元素的概述，例如主要的结构性元素、重要的行为和具有重大影响的决策	首席架构师
架构概念证明	用于证明和确认架构的关键部分以降低项目的风险。这个工作产品可能采用统一建模语言（UML）的解决方案模型或一个模拟的解决方案，或一个可执行的原型	首席架构师

（续）

工作产品名称	描　　述	拥有的角色
业务实体模型	定义了考虑的业务领域中的关键概念	业务分析人员
业务流程模型	定义了由业务和执行每个活动的角色所进行的活动	业务分析人员
业务规则	定义了开发中系统必须满足的所有政策或条件	业务分析人员
变更请求	描述了请求改变系统的要求。这些改变可能很重大并影响迭代的关注点	项目经理
数据模型	描述系统使用的数据的呈现方式	数据架构师
部署模型	列出了节点的配置、节点之间的通信和部署在节点上的组件	基础设施架构师
企业架构规范	包含了定义的企业级规则和方针，告知并指导架构创建的方式	首席架构师
现有 IT 环境	代表一组组成当前 IT 环境的现有元素，开发中的系统可能使用它们或受其约束	首席架构师
功能模型	描述软件组件，包括它们的功能和相互之间的关系，以及在提供要求的功能时组件之间的协作	应用架构师
功能性需求	描述开发中系统的功能性需求	业务分析人员
术语表	常用术语的清单、它们的含义及业务中可替代的词	业务分析人员
非功能性需求	描述了开发中系统的非功能性需求。非功能性需求表示系统的质量（如可靠性）及对它的限制	业务分析人员
排定优先级的需求清单	定义每项需求的优先级	业务分析人员
RAID 日志	获取项目的风险、假设、问题和依赖条件	业务分析人员
复审记录	记录复审过程中的重要发现以及处理那些发现的推荐行动	项目经理
软件架构文档	通过大量架构视图来描述系统的不同方面，从而提供一个全面的系统架构概述	首席架构师
利益相关者请求	包含了所开发系统的利益相关者可能会提出的要求。它可能还包括系统必须遵循的参考标准（如服从监管的标准或组织的标准）	业务分析人员
系统上下文	把系统描述为一个单一的实体或过程并确认系统和外部实体之间的接口	业务分析人员
愿景	定义利益相关者对所开发系统的展望并明确说明利益相关者的关键需求和特性。它概述了所设想的系统的核心需求，可能就像一个问题说明一样简单	业务分析人员

图 C.2 显示了工作产品和视点之间的对应关系。

	需求视点	功能性视点	部署视点	验证视点
应用视点	业务实体模型 业务流程模型 业务规则 变更请求 企业架构规则 现有 IT 环境 功能性需求 术语表 非功能性需求 排定优先级的需求列表 利益相关者请求 系统背景 愿景	架构决策 架构概览 数据模型 功能性模型	架构决策 架构概览 部署模型	架构评估 架构概念证明（风险、假设、问题和依赖条件）日志 复审日志
基础结构视点				
系统管理视点				
可用性视点				
性能视点				
安全视点				

图 C.2　视点和工作产品

活动

一个活动代表一组任务。图 C.3 中描述的活动遵循第 7 章、第 8 章和第 9 章描述的流程。另外，这个图还显示了在整个开发过程中可能采取的更详细的设计活动，虽然这些活动在本书中没有进行详细的讨论。您应该记住，在一个迭代的开发过程中，这些活动的每一个都会在单次迭代中执行。

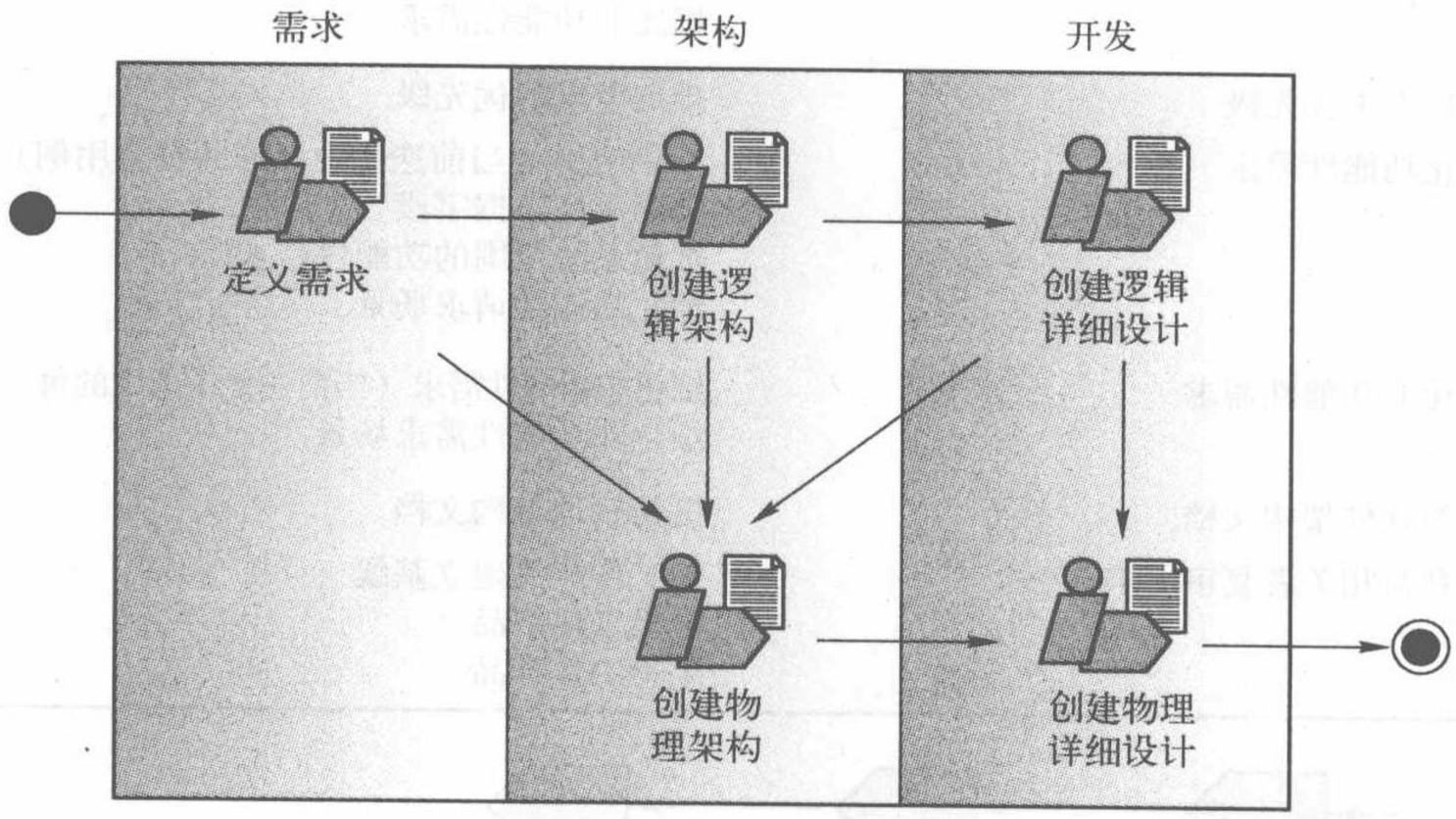


图 C.3　活动概览

任务

任务是提供一个在项目的上下文中有意义的结果的工作单元。它有清楚的目的，通常涉及创建或更新工件。在接下来的部分，各种各样的任务根据它们所属的活动组织在一起。每个活动都有一个显示任务之间流程的整体的活动图，还有一个概述每个任务步骤的表格。

活动：定义需求

在组成**定义需求**活动的这些任务之间的流程如图 C.4 所示，表 C.3 中概述了其中的每一个任务。

表 C.3　定义需求：任务概述

任务名称	步　骤
收集利益相关者请求	确定利益相关者 收集利益相关者请求 排定利益相关者请求的优先级
获取常用词汇	确定常用术语

（续）

任务名称	步　骤
定义系统上下文	确定参与者 确定参与者位置 确定数据流
概述功能性需求	确定功能性需求 概述功能性需求
概述非功能性需求	确定非功能性需求 概述非功能性需求
排定需求优先级	排定需求的优先级
细化功能性需求	细化用例（当前迭代中考虑的每个用例） 细化用例数据字段 细化系统范围的功能性需求 细化功能性请求场景
细化非功能性需求	细化非功能性需求（当前迭代中考虑的每一个非功能性需求） 细化非功能性需求场景
更新软件架构文档	更新软件架构文档
与利益相关者复审需求	为工作产品建立基线 汇集工作产品 复审工作产品

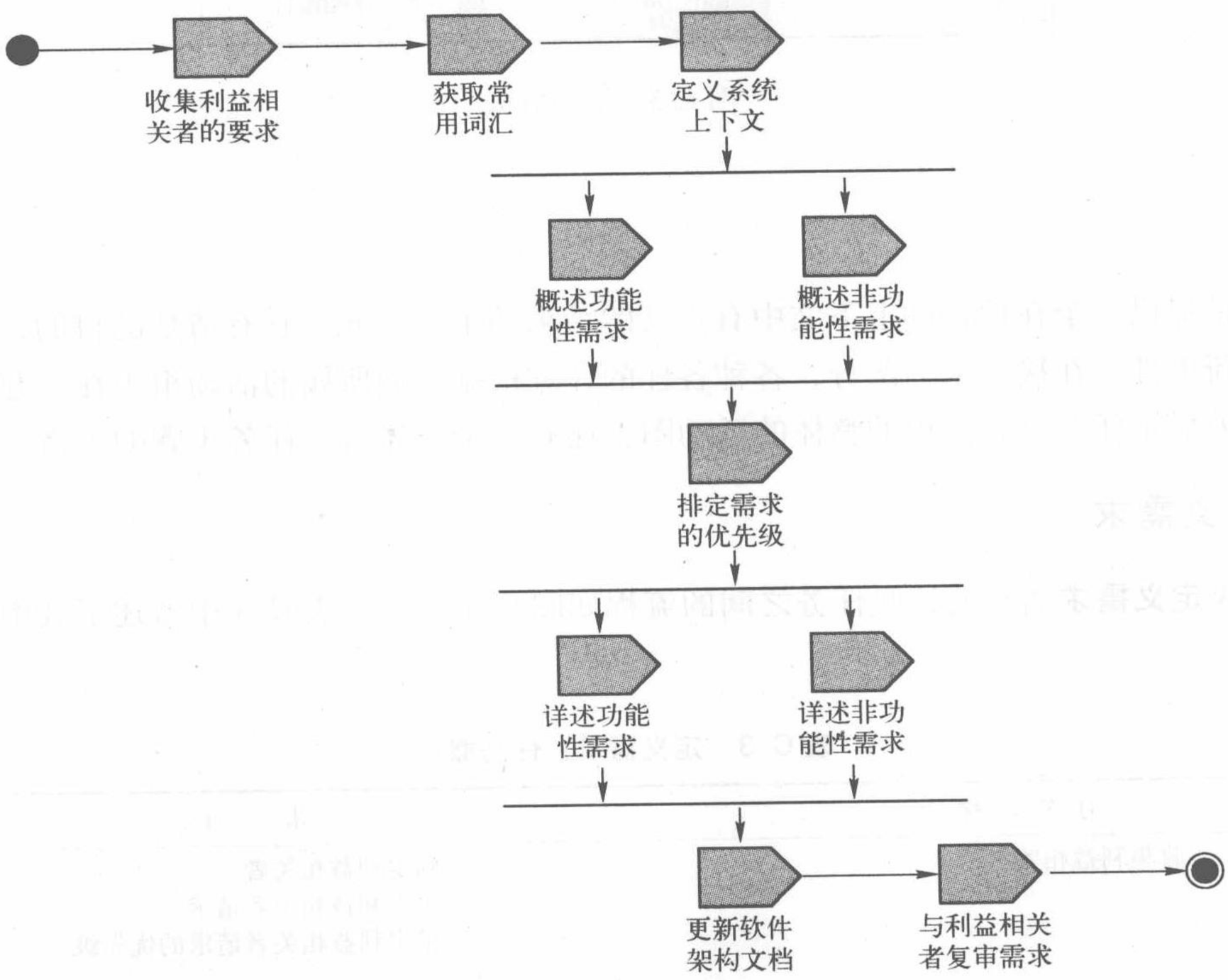


图 C.4　定义需求活动中的任务

活动：创建逻辑架构

在组成**创建逻辑架构**活动的这些任务之间的流程如图 C.5 所示，表 C.4 中概述了其中的每一个任务。

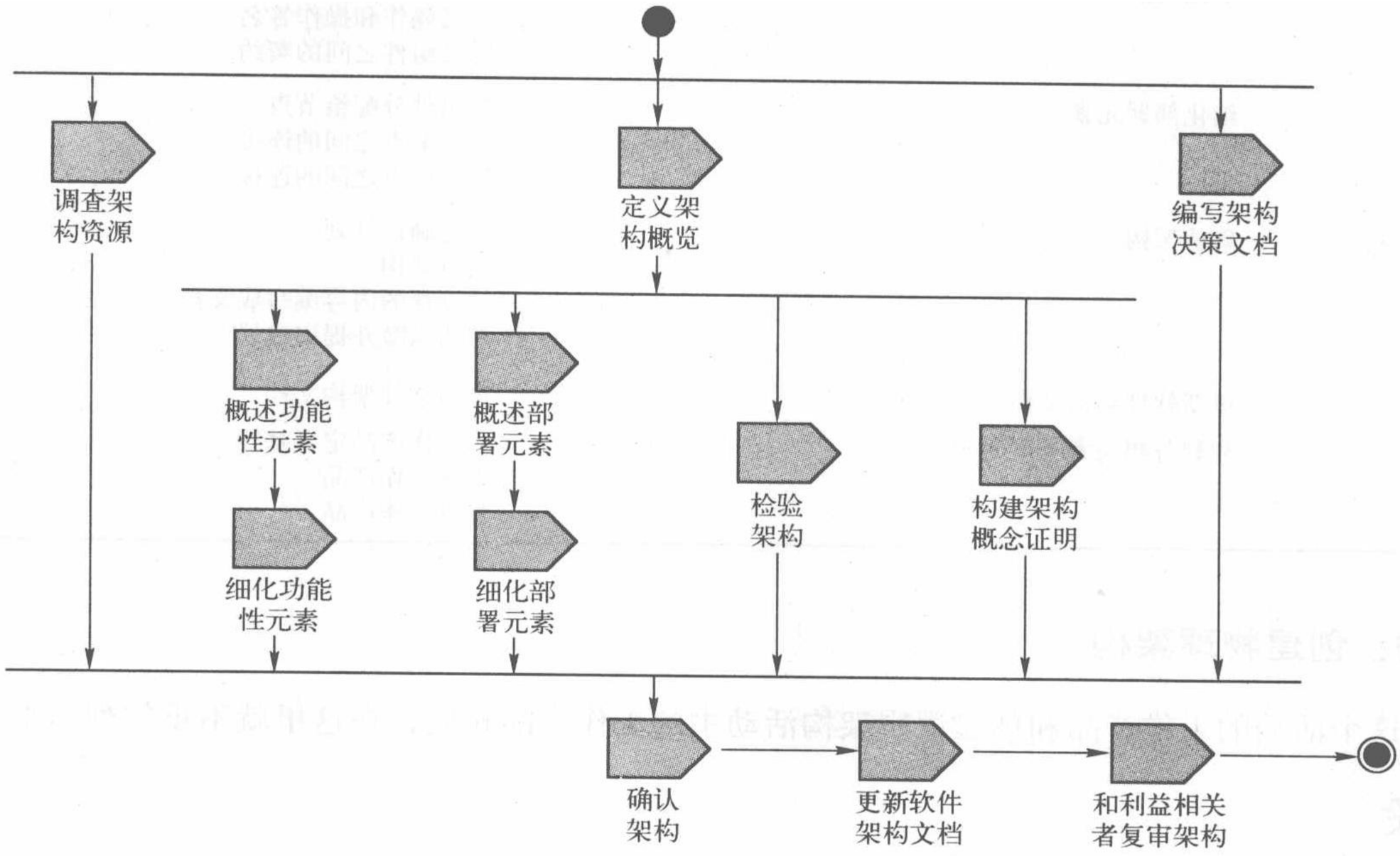


图 C.5　创建逻辑架构活动中的任务

表 C.4　创建逻辑架构：任务概述

任务名称	步　骤
调查架构资源	调查架构资源
定义架构概览	定义架构概览
编写架构决策文档	捕获问题或关注点 评估选项 选择最优选项 编写决策文档
概述功能性元素	确定子系统 确定组件
概述部署元素	确定位置 确定节点
检验架构	计划检验 召开动员会议 进行个别的检验 召开检验会议 进行返工 进行后续工作

（续）

任务名称	步骤
构建架构概念证明	创建架构的概念证明 把发现的内容编写成文档
细化功能性元素	定义组件接口 定义操作和操作签名 定义组件之间的契约
细化部署元素	把组件分配给节点 定义节点之间的连接 定义地点之间的连接
确认架构	制定确认计划 复审架构 把发现的内容编写成文档 评估风险并提出建议
更新软件架构文档 和利益相关者复审架构	更新软件架构文档 为工作产品定义基线 聚集工作产品 复审工作产品

活动：创建物理架构

这个活动的工作产品和**创建逻辑架构**活动中的工作产品相同，在这里就不重复列出来了。

阶段

阶段是一个专门的活动类型，它代表项目中以一个决策点、主要里程碑或一组可交付物结束的一个重要时期。阶段通常拥有良好定义的目标并为如何组织项目工作提供基础。（OpenUP 2008）

起始

起始阶段是确定项目的业务案例及确定利益相关者对项目的目标达成一致的时间段。在起始阶段，架构师的关注点是确保项目既有价值又切实可行。表 C.5 概述了起始阶段的目标和里程碑评估标准。

表 C.5　起始阶段概述

主要目标	里程碑评估标准
建立项目范围	利益相关者同意这个范围定义
建立项目的接收标准	利益相关者同意这个项目的接收标准
确定系统的特性并选择关键的特性	利益相关者同意已经获得恰当的需求并对需求有了共同的理解。所有的需求都排了优先级
评估整个项目的总成本和进度（细化阶段的更详细的评估会立刻跟进）	利益相关者同意成本/进度的评估、优先级、风险和开发流程都是适当的

（续）

主要目标	里程碑评估标准
评估潜在的风险	所有已知的风险都已经记录并进行了评估，降低风险的策略也已经确定
建立项目的支持环境（如硬件、软件、流程和资源）	支持环境准备完毕

细化

细化阶段是建立架构以便为在构造阶段执行的活动提供稳定基础的时间段。因此，这个阶段尤其与架构师相关，当然，在这个阶段，架构师需要消耗最多的精力。表 C.6 概述了细化阶段的目标和里程碑评估标准。

表 C.6 细化阶段概述

主要目标	里程碑评估标准
确保架构、需求和计划稳定，建立一个基础的架构	产品愿景、需求和基础的架构稳定
确保风险充分降低以可预测地确定开发完成的成本和计划	主要的风险因素已经处理并可靠地解决
证明基础架构能在可接受的成本和可接受的时间范围内支持系统的需求	系统和所选功能范围中的架构上的所有重要方面都已经在架构概念证明中获得评估

构造

构造阶段是澄清遗留需求和基于细化阶段确定的基础架构完成系统开发的时间段。在细化和构造阶段之间，关注点从理解问题和确定解决方案的关键元素转移到开发一个可部署产品。表 C.7 概述了构造阶段的目标和里程碑评估标准。

表 C.7 构造阶段概述

主要目标	里程碑评估标准
及时地完成有用的版本（内部测试版本、外部测试版本、其他测试版本）	系统按阶段计划和迭代计划指定的预期进度及时开发
完成所有必需功能的分析、设计、实现和测试	所有必需的功能都合并到系统中
确定系统是否准备好部署	在开发环境中测试时，系统已经达到所有接受的标准

移交

移交阶段是架构师确保这个软件对于最终用户可用和可接受的时间段。就是在这个阶段，系统部署到用户的环境以评估和测试。关注点在于调整这个产品和解决配置、安装和可用性的问题。在移交阶段末期，这个项目应该能够停止。表 C.8 概述了移交阶段的目标和里程碑评估标准。

表 C.8 移交阶段概述

主要目标	里程碑评估标准
成功地把系统进行到交付的轨道上	系统通过了部署环境中正式的验收标准

附录 D

架构需求检查列表

这个附录包括一个关于有特定架构意义的潜在需求的检查列表。例如，在收集利益相关者请求和复审需求时可以使用这样的一个检查列表。

有好几个需求类型分级。其中一个分级是 FURPS 分类（Grady 1992），字母分别代表功能性（functionality）、可用性（usability）、可靠性（reliability）、性能（performance）和支持性（supportability）。在 ISO 标准 9126（ISO 9126 2001）中也定义了一个类似的分级，其中，需求分为外部和内部的质量以及使用的质量。在这个标准中，“外部和内部的质量特性”是功能性、可靠性、可用性、效率、可维护性和可移植性。“使用的质量特性”是效率、生产率、安全和满意度。《Software Architecture in Practice，2nd ed》（Bass 2003）中也列出了各种非功能性需求的特性，其中，作者们考虑系统质量、业务质量和架构质量本身。

本书中使用的分级是建立在 FURPS 分级的基础之上的，尤其强调非功能性需求（FURPS 中的 URPS）。在这个附录中，我们对这个分级进行扩展，把约束也强调为检查列表的关键部分（我们称之为 FURPS + ）。我们鼓励您采用这样一个检查列表，根据需要进行调整并把它应用到您自己的项目中。

功能性需求

功能性需求描述系统在定义的条件下使用时应该展示的功能。

功能性需求通常是开发中系统所特有的，所以，我们没有包括一个应该考虑的所有功能性需求的检查清单。相反，本书建议基于与系统交互的参与者的需要，使用用例建模的技术来获取大部分的功能性需求。

有一些功能性需求通过用例来获取并不是最好的，在本书中，这些功能称为系统范围的功能性需求（从某种意义上来说，它们并不与某一特定用例有关）。例如，提供一个安全的系统，这是一个遍及整个系统的功能性需求，并不依赖某一特定用例。表 D.1 提供了一些出现在系统范围内且架构师特别感兴趣的功能性需求的例子。

表 D.1　系统范围的功能性需求的例子

需　　求	描　　述
审计	提供审计系统的能力
本地化	提供本地化机制以允许系统配置为使用某一特定人类语言
在线帮助	提供在线帮助的能力
打印	提供打印的能力

（续）

需　求	描　述
报告	提供报告的能力
安全	提供阻止资源不遭受未授权访问的安全能力
系统管理	提供系统管理能力来控制系统的运转特性

可用性需求

可用性需求描述了对系统的理解和使用的程度。表 D.2 定义了一些可用性需求。

表 D.2　可用性需求的例子

需　求	描　述
可达性	系统通过特定的用户交互机制进行使用的难易程度
美观性	系统对用户的吸引程度
可学性	用户学习系统的难易程度
可操作性	用户操作系统的难易程度

可靠性需求

可靠性需求描述了系统提供定义的操作性能的程度。表 D.3 定义了一些可靠性需求。

表 D.3　可靠性需求的例子

需　求	描　述
准确性	系统提供结果的精确程度
有效性	系统保持特定运行状态的难易程度
可恢复性	系统重新建立某一特定运行状态的难易程度

性能需求

性能需求描述了系统提供规定的执行性能的程度。表 D.4 定义了一些性能需求。

表 D.4　性能需求的例子

需　求	描　述
资源消耗	系统对规定资源的消耗程度
速度	系统展现的特定处理时间（如启动、关闭和响应时间）的长短程度
生产能力	系统对指定生产能力的支持程度

支持性需求

支持性需求描述了系统可以支持的程度。表 D.5 定义了一些支持性需求。

表 D.5 支持性需求的例子

需　求	描　述
适应性	系统对新环境的适应程度
兼容性	系统对先前的或将来的版本的兼容程度
可配置性	系统配置的难易程度
可扩展性	系统扩展的难易程度
可安装性	系统安装的难易程度
协同性	系统与其他特定系统交互的难易程度
可维护性	系统修改的难易程度
可替换性	系统用来替换相同环境中担任相同任务的其他系统的能力，包括替换系统先前的或将来的版本
可量测性	系统在数据容量和并行用户方面的可量测性
可测试性	系统验证的难易程度

约束

约束是我们在提供解决方案的过程中对自由程度的限制。（Leffingwell 2000）

本质上，约束表示对解决方案的限制，因为它们限制了可用的选项。

业务约束

影响架构的一些约束可能来自业务。表 D.6 定义了一些与业务相关的约束。

表 D.6 业务约束的例子

需　求	描　述
顺从	坚持要求的标准和规则的能力
成本	解决方案和其部署的成本的约束
进度	对开发和部署解决方案的进度的约束

架构约束

各种各样的架构机制可能受到约束。表 D.7 定义了一些影响架构的约束。当然，更多的机制在任意情况下都能起作用。

表 D.7 架构约束的例子

需　求	描　述
协作	对处理用户之间协作的机制的约束
通信	对处理进程间通信的机制的约束
错误管理	对管理系统内错误的机制的约束
事件管理	对处理系统内异步事件的机制的约束
集成	对允许系统和其他系统集成的机制的约束

（续）

需　求	描　述
持久	对允许系统数据写到磁盘上的机制的约束
调度	对提供调度能力的机制的约束
安全	对提供安全能力的机制的约束
系统管理	对提供系统管理能力的机制的约束
事务管理	对处理事务的机制的约束
工作流	对处理工作项移动（通常经由一个组织）的机制的约束

开发约束

另外的约束可能影响详细设计和开发。表 D.8 定义了一些影响开发和运行期环境的约束。

表 D.8　开发约束的例子

需　求	描　述
实现语言	使用的编程语言的约束
接口格式	在当前系统和外部系统之间传输的数据格式的约束
遗留集成	使用现有组件的约束
网络基础设施	系统使用的网络基础设施的约束
平台支持	系统将支持的平台的约束
资源限制	受资源使用的任何限制所影响的约束，如内存和硬盘空间
标准顺应性	受系统必须遵守的任何标准所影响的约束
第三方组件	第三方组件的使用和成本的约束

物理约束

表 D.9 定义了一些受物理环境影响的约束。当承载软件的硬件本身在某些方面受到约束时，这些需求尤其能造成影响。便携式电话、飞机和军事硬件（如坦克）都对计算机硬件有物理上的约束。例如，这些约束本身都可能影响在这些硬件上执行的软件的约束。

表 D.9　物理约束的例子

需　求	描　述
功耗	对系统功耗的约束
形状	对最终承载系统的硬件的形状的约束
大小	对承载系统的硬件的大小的约束
重量	对承载系统的硬件的重量的约束

术　语　表

这个术语表定义了本书中使用的术语。

activity（活动）：一组任务。

antipattern（反模式）：一种普遍存在的形成决定性的否定逻辑关系的模式或解决方案。反模式可能是一个位于错误上下文中的模式。（Brown 1998）

application framework（应用程序框架）：一个应用程序的特定方面的部分实现。

architect（架构师）：负责系统架构的人、团队或组织。（IEEE 1471 2000）这个术语等同于**软件架构师**。

architecting（架构设计）：一个架构的定义、文档编写、维护、改进和验证正确实现的活动。（IEEE 1471 2000）这个术语等同于**软件架构设计**。

architectural description（架构描述）：描述一个架构的文档集。（IEEE 1471 2000）

architectural mechanism（架构机制）：对经常遭遇到的问题的共同的具体解决方案。它们可能是结构模式、行为模式或两者都是。（SPEM 2007）

architectural style（架构风格）：根据结构组织的模式定义系统种类。更具体地说，架构风格定义组件和连接器类型的词汇及它们如何进行组合的一组约束。（Shaw 1996）

architecture（架构）：体现在它的组件中的一个系统的基本组织、它们彼此的关系、与环境的关系及指导它的设计和发展的原则。（IEEE 1471 2000）

architecture decision（架构决策）：关于一个软件系统整体或它的一个或多个核心组件的刻意设计决策。这些决策决定系统的非功能性特性和质量指标。（Zimmermann 2008）

business rule（业务规则）：定义或约束业务某些方面的声明。它的意图是为了确定业务结构或为了控制或影响业务的行为。（业务规则组 2000）

component（组件）：代表一个系统模块化的一部分，它封装了自身的内容，在环境中它的表现是可替换的。一个组件根据规定接口和要求接口定义自己的行为。同样，一个组件也属于某一个类型，其一致性由这些规定接口和要求接口（包括它们的静态语义和动态语义）来定义。因此，一个组件可以由另一个组件替换，只要这两个组件类型一致。（UML 2.2 2009）

concern（关注点）：与系统开发、系统操作或其他对多个利益相关者重要或关键的方面相关的兴趣点。关注点包括像性能、可靠性、安全、分布方式和可扩展性这样的系统考虑。（IEEE 1471 2000）

constraint（约束）：对我们提供的解决方案的自由程度的一个限制。（Leffingwell 2000）

design authority（设计权威）：专门负责系统的整体技术完整性和提供把开发中的 IT 系统和企业架构目标、战略、策略联系起来的必要控制以确保系统满足要求的角色。

development process（开发流程）：在系统开发过程中为了特定目标而执行的一组部分排序的

步骤，例如构建模式或实现模式。（UML 2.0 2003）

development project（开发项目）：一个承诺创建一个独特产品或服务的临时性意图。临时性的意思是每个项目都有一个明确的起点和一个明确的终点。独特的意思是这个产品或服务和所有类似的产品或服务存在差异。项目通常是进行组织业务战略的关键组成部分。（RUP 2008）

discipline（规范）：把定义一个主要“关注范围”和/或协作工作成果的任务组织在一起的一个主要分类机制。（OpenUP 2008）

domain（领域）：从事于某一行业的人理解的归纳为一组概念和术语的知识或活动范围。（UML User Guide 1999）

enterprise architecture（企业架构）：当与业务战略和信息需求保持一致时，指导与将来的业务方向保持一致的解决方案的选择、创建和实现的一组原则、指导、政策、模型、标准和流程。

environment（环境）（或上下文）：决定对系统的开发性、操作性、政策性或其他方面产生影响的情况和设置。（IEEE 1471 2000）

functional requirement（功能性需求）：支持用户目标、任务或活动的 IT 系统的行为（功能或服务）描述。（Malan 2001）

iteration（迭代）：一个项目的短的周期划分。迭代使您能够逐步证明价值并及早地获得持续反馈。（OpenUP 2008）

method（方法）：实现某些东西的有规则的和系统的方式；完成一个任务或达到一个目标所遵循的详细的、有逻辑的和有序的计划或过程。（RUP 2008）

mission（目标）：一个或多个利益相关者通过使用系统来达到的一组目标。（IEEE 1471 2000）

model（模型）：一个架构的特定说明或内容。例如，一个结构性视图可能由一组系统结构模型组成。这些模型的元素可能包括可确认的系统组件和它们的界面，以及在这些组件之间的联系。（IEEE 1471 2000）

non-functional requirement（非功能性需求）：包含约束和质量的需求。（Malan 2001）

packaged application（封装的应用程序）：提供很多能力（和重用）的一个粗粒度的商业化成熟（COTS）产品，例如一个客户关系管理（CRM）应用程序或企业资源计划（ERP）应用程序。

pattern（模式）：在特定上下文中常见问题的一个常规解决方案。（UML User Guide 1999）

phase（阶段）：一个专门的活动类型，它代表项目中以一个决策点、主要里程碑或一组可交付物结束的一个重要时期。阶段通常拥有良好定义的目标并为如何组织项目工作提供基础。（OpenUP 2008）

practice（实践）：解决一个或者多个经常发生的问题的方法。这些方法也被有意作为采用、启动和配置的流程“块”（chunk）。（RUP 2008）

process（流程）：参考开发流程。

quality（质量）：系统的利益相关者所关心的、会影响其对系统的满意度的系统的属性或特

征。(Malan 2001)

rationale (基本原理)：一系列行动或一个信念的一组理由或逻辑基础。(OED)

reference architecture (参考架构)：与特定的兴趣领域相关的架构，它通常包括许多不同的架构模式、应用在它的结构的不同范围。

reference model (参考模型)：是某一特定兴趣领域中实体、它们的关系和行为的一个抽象表示，它通常形成开发比较具体的元素的概念基础。例子包括业务模型、信息模型、术语表等。

reusable asset (可重用资源)：重复出现的问题的解决方案。可重用资源是已经开发过的在头脑中重用的一个资源。(RAS 2004)

role (角色)：在软件开发组织的上下文中，一个人或作为一个团队在一起工作的一组人的职责。一个角色负责一个或多个工作产品并执行一组任务。

stakeholder (利益相关者)：是对系统感兴趣的或与系统有关系的一个单独的团队或组织（或组织的一部分)。(IEEE 1471 2000)

subsystem (子系统)：一组相关的组件。

system (系统)：(1) 组织在一起来实现一个或一组功能的组件集合。系统这个术语包括单独的应用程序、传统意义上的系统、子系统、系统之系统、产品线、产品族、整个企业及其他利益组合。一个系统存在以在它的环境中完成一个或多个任务。(IEEE 1471 2000)

(2) 由一个或多个流程、硬件、软件、设施和人员组成的提供一个能力来满足一个特定需要或目标的一个完整的复合物。(IEEE 12207 1995)

task (任务)：在项目的上下文中提供有意义结果的一个工作单元。它有明确的目的，通常涉及创建或更新工作产品。所有的任务都由适当的角色执行。

team (团队)：拥有共同目的、执行目标、拥有使他们可以相互负责的方法的、技能相互补充的小部分人。(Katzenbach 1993)

traceability (可追溯性)：我们可以用来从一组开发元素追踪（确认和衡量）到另一组开发元素的机制。

validation (确认)：关注整个系统而不是个别工作产品的一个流程。它关注于确保系统满足规定的需求。

verification (检验)：检验使我们能够判断当前的工作产品在整个开发进程中是否和为创建它们而使用的输入工作产品一致（例如其他工作产品和任何强制的开发标准)。

view (视图)：把整个系统看作是一组相关关注点的一种表示法。(IEEE 1471 2000)

viewpoint (视点)：构建和使用一个视图的一个传统规范。通过确定视图目标和拥护者及确定创建和分析该视图所用的技术来开发不同视图的模式或模板。(IEEE 1471 2000)

viewpoint catalog (视点目录)：一组视点的集合。

work product (工作产品)：在流程执行过程中生成或使用的一组信息或物理实体。工作产品的例子包括模型、计划、代码、可执行程序、文档、数据库等。

参 考 文 献

Agile Manifesto 2009. "The Agile Manifesto." www.agilemanifesto.org (accessed April 2, 2009).

Ahern 2008. Ahern, Dennis M., Aaron Clouse, and Richard Turner. 2008. *CMMI Distilled:A Practical Introduction to Integrated Process Improvement*, 3rd ed. Boston: Addison-Wesley.

Alexander 1964. Alexander, Christopher. 1964. *Notes on the Synthesis of Form*. Cambridge, MA: Harvard University Press.

Alexander 1977. Alexander, Christopher, Sara Ishikawa, and Murray Silverstein. 1977. *A Pattern Language*. New York: Oxford University Press.

Alexander 1979. Alexander, Christopher. 1979. *The Timeless Way of Building*. New York: Oxford University Press.

Ambler 2000. Ambler, Scott W. 2000. *The Unified Process Elaboration Phase*. Lawrence, KS: R&D Books.

Ambler 2008. Ambler, Scott, and Celso Gonzalez. 2008. *"Agile Model-Driven Development."* www.stickyminds.com/BetterSoftware/magazine.asp?fn=cifea&ac=367 (accessed April 4, 2009).

Barbacci 2003. Barbacci, Mario R., Robert Ellison, Anthony J. Lattanze, Judith A. Stafford, Charles B. Weinstock, and William G. Wood. 2003. "Quality Attribute Workshops (QAWs)," 3rd ed. www.sei.cmu.edu/publications/documents/03.reports/03tr016.html (accessed April 4, 2009).

Bass 2003. Bass, Len, Paul Clements, and Rick Kazman. 2003. *Software Architecture in Practice*, 2nd ed. Boston: Addison-Wesley.

Bittner 2003. Bittner, Kurt, and Ian Spence. 2003. *Use Case Modeling*. Boston: Addison-Wesley.

Booch 2009. Booch, Grady. 2009. "Handbook of Software Architecture," www.handbookofsoftwarearchitecture.com (accessed April 4, 2009).

Bosch 2000. Bosch, Jan. 2000. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Harlow, England: Addison-Wesley.

Brown 1998. Brown, William, Raphael Malveau, Hays McCormick and Thomas Mowbray. 1998. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. New York: John Wiley and Sons.

Buschmann 1996. Buschmann, Frank, et al. 1996. *Pattern-Oriented Software Architecture:A System of Patterns*. Chichester, England: John Wiley and Sons.

Buschmann 2009. Buschmann, Frank, and Christa Schwanninger. 2099. "Architecture Reviews." Frank Presentation at OOP, Munich, Germany, 2009.

Business Rules Group 2000. The Business Rules Group. 2000. "Defining Business Rules: What Are They Really?" www.businessrulesgroup.org (accessed April 4, 2009).

Cantor 2003. Cantor, Murray. 2003. "Rational Unified Process for Systems Engineering." *The Rational Edge*. http://download.boulder.ibm.com/ibmdl/pub/software/dw/rationaledge/aug03/f_rupse_mc.pdf (accessed April 4, 2009).

Cheesman 2001. Cheesman, John, and John Daniels. 2001. *UML Components: A Simple Process for Specifying Component-Based Software*. Boston: Addison-Wesley.

Clements 2001. Clements, Paul, and Linda Northrop. 2001. *Software Product Lines: Practices and Patterns*. Boston: Addison-Wesley.

Clements 2002. Clements, Paul, Rick Kazman, and Mark Klein. 2002. *Evaluating Software Architectures: Methods and Case Studies*. Boston: Addison-Wesley.

Clements 2003. Clements, Paul, et al. 2003. *Documenting Software Architectures: Views and Beyond*. Boston: Addison-Wesley.

Cockburn 2000. Cockburn, Alistair. 2000. *Writing Effective Use Cases*. Boston: Addison-Wesley.

Conallen 2003. Conallen, Jim. 2003. *Building Web Applications with UML*, 2nd ed. Boston: Addison-Wesley.

Coplien 2005. Coplien, James, and Neil Harrison. 2005. *Organizational Patterns of Agile Software Development*. Upper Saddle River, NJ: Prentice Hall.

Eeles 2008. Eeles, Peter. 2008. "The Rise of the Development Environment Architect." *The Rational Edge*. www.ibm.com/developerworks/rational/library/edge/08/apr08/eeles/index.html (accessed April 4, 2009).

Fagan 1976. Fagan, Michael. 1976. "Design and code inspections to reduce errors in program development." *IBM Systems Journal* 15:3;182–211.

Fowler 1997. Fowler, Martin. 1997. *Analysis Patterns: Reusable Object Models*. Menlo Park, CA: Addison-Wesley.

Gamma 1995. Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley.

Gilb 1988. Gilb, Tom. 1988. *Principles of Software Engineering Management*. Wokingham, England: Addison-Wesley.

Gilb 1993. Gilb, Tom, and Dorothy Graham. 1993. *Software Inspection*. Wokingham, England: Addison-Wesley.

Grady 1992. Grady, Robert B. 1992. *Practical Software Metrics for Project Management and Process Improvement*. Upper Saddle River, NJ: Prentice Hall.

Herzum 2000. Herzum, Peter, and Oliver Sims. 2000. *Business Component Factory*. New York: John Wiley and Sons.

Hofmeister 2000. Hofmeister, Christine, Robert Nord, and Dilip Soni. 2000. *Applied Software Architecture*. Reading, MA: Addison-Wesley.

Hofmeister 2005: Hofmeister, Christine et al. 2005. "Generalizing a Model of Software Architecture Design from Five Industrial Approaches." Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA5). Washington, DC: IEEE Computer Society.

Hopkins 2008. Hopkins, Richard, and Kevin Jenkins. 2008. *Eating the IT Elephant: Moving from Greenfield Development to Brownfield*. Upper Saddle River, NJ: IBM Press.

IAA 2009. IBM. "Insurance Application Architecture." www-03.ibm.com/industries/insurance/us/detail/solution/P669447B27619A15.html (accessed April 4, 2009).

IBM 2009. "IBM patterns for e-business." www.ibm.com/developerworks/patterns/ (accessed April 4, 2009).

IEEE 1061 1992. IEEE Computer Society. 1992. Standard for a Software Quality Metrics Methodology. IEEE Std 1061-1992. http://standards.ieee.org/reading/ieee/std_public/description/se/1061-1992_desc.html (accessed April 4, 2009).

IEEE 12207 1995. IEEE Computer Society. 1995. IEEE Standard for Information Technology—Software Life Cycle Processes. IEEE Std 12207-1995. http://ieeexplore.ieee.org/ISOL/standardstoc.jsp?punumber=5410 (accessed April 4, 2009).

IEEE 1471 2000. IEEE Computer Society. 2000. Recommended practice for architectural description of software-intensive systems. IEEE Std 1471-2000. http://standards.ieee.org/reading/ieee/std_public/description/se/1471-2000_desc.html (accessed April 4, 2009).

ISO 9126 2001. International Organization for Standardization/International Electrotechnical Commission. 2001. ISO/IEC 9126-1:2001. Software engineering—Product quality—Part 1: Quality model. www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=22749 (accessed April 4, 2009).

Jacobson 1992. Jacobson, Ivar, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard. 1992. *Object-Oriented Software Engineering:A Use Case Driven Approach*. Wokingham, England: Addison-Wesley.

Jacobson 1997. Jacobson, Ivar, Martin Griss, and Patrik Jonsson. 1997. *Software Reuse:Architecture, Process and Organization for Business Success*. Harlow, England: Addison-Wesley.

Katzenbach 1993. Katzenbach, Jon R., and Douglas K. Smith. 1993. *The Wisdom of Teams*. Boston: Harvard Business School Press.

Kruchten 1995. Kruchten, Philippe. 1995. "The '4+1' View Model of Software Architecture." *IEEE Software* 12:6;42–50.

Kruchten 1995-2. Kruchten, Philippe. 1995. "Mommy, Where Do Software Architectures Come From?" Proceedings of 1st International Workshop on Architectures for Software Systems, Seattle, 1995.

Kruchten 1999. Kruchten, Philippe. 1999. "The Architects: The Software Architecture Team." Donohoe, Patrick, ed. *Proceedings of the First Working IFIP Conference on Software Architecture* (WICSA1). Deventer, Netherlands: Kluwer Academic Publishing.

Kruchten 2000. Kruchten, Philippe. 2000. *The Rational Unified Process:An Introduction*, 2nd ed. Reading, ME: Addison-Wesley.

Leffingwell 2000. Leffingwell, Dean, and Don Widrig. 2000. *Managing Software Requirements:A Unified Approach*. Reading, MA: Addison-Wesley.

Malan 2001. Malan, Ruth, and Dana Bredemeyer. 2001. "Defining Non-Functional Requirements." www.bredemeyer.com/pdf_files/NonFunctReq.PDF (accessed April 4, 2009).

Marasco 2004. Marasco, Joe. 2004. "On Politics in Technical Organizations." *The Rational Edge*. www.ibm.com/developerworks/rational/library/4690.html (accessed April 4, 2009).

McGovern 2004. McGovern, James, Scott W. Ambler, Michael E. Stevens, James Linn, Vikas Sharan, and Elias K. Jo. 2004. *A Practical Guide to Enterprise Architecture*. Upper Saddle River, NJ: Prentice Hall.

MDA 2009. Object Management Group. 2009. "Model Driven Architecture." www.omg.org/mda/ (accessed April 4, 2009).

MDSD 2009. IBM Redbooks. 2009. "Model Driven Systems Development with Rational Products." www.redbooks.ibm.com/abstracts/sg247368.html

(accessed April 4, 2009).

Meyer 1997. Meyer, Bertrand. 1997. *Object-Oriented Software Construction*, 2nd ed. Upper Saddle River, NJ: Prentice-Hall.

Microsoft 2009. "Microsoft Patterns and Practices Developer Center." http://msdn.microsoft.com/en-gb/practices/default.aspx (accessed April 4, 2009).

OED 2009. AskOxford.com. www.askoxford.com (accessed April 4, 2009).

OpenUP 2008. Open Unified Process, Version 1.5.0.1. 2008. http://epf.eclipse.org/wikis/openup/ (accessed April 4, 2009).

RAS 2004. Reusable Asset Specification. Object Management Group, Inc. Document number 04-06-06. www.omg.org/cgi-bin/doc?ptc/2004-06-06 (accessed April 4, 2009).

Royce 1998. Royce, Walker. 1998. *Software Project Management:A Unified Framework*. Upper Saddle River, NJ: Addison-Wesley.

Royce 2005. Royce, Walker. 2005. "Successful software management style: Steering and balance." *The Rational Edge*. www.ibm.com/developerworks/rational/library/mar05/royce/ (accessed April 4, 2009).

Rozanski 2005. Rozanski, Nick, and Eoin Woods. 2005. *Software Systems Architecture:Working with Stakeholders Using Viewpoints and Perspectives*. Upper Saddle River, NJ: Addison-Wesley.

RUP 2008. Rational Unified Process. IBM Rational Software, Version 7.5. www.ibm.com/software/awdtools/rmc (accessed April 4, 2009).

SARA 2002. Software Architecture Review and Assessment (SARA) Report, Version 1.0. Henk Obbink, Henk, et al. 2002. http://philippe.kruchten.com/architecture/SARAv1.pdf (accessed April 4, 2009).

Schwaber 2002. Schwaber, Ken, and Mike Beedle. 2002. *Agile Software Development with Scrum*. Upper Saddle River, NJ: Prentice Hall.

SEI 2009. Software Engineering Institute (SEI) Architecture website. www.sei.cmu.edu/architecture (accessed April 4, 2009).

Shaw 1996. Shaw, Mary, and David Garlan. 1996. *Software Architecture: Perspectives on an Emerging Discipline*. Upper Saddle River, NJ: Prentice Hall.

SPEM 2007. Software & Systems Process Engineering Metamodel Specification, Version 2.0. Object Management Group, Inc. Document number 07-08-07, October 2007. www.omg.org/technology/documents/formal/spem.htm (accessed April 4, 2009).

Stevens 1974. W. P. Stevens, G. J. Myers, L. L. Constantine. 1974. "Structured design." *IBM Systems Journal* 13:2;115.

Sun 2009. Sun Java BluePrints. http://java.sun.com/blueprints/patterns/index.html (accessed April 4, 2009).

TOGAF 2009. The Open Group Architecture Framework. TOGAF 8.1.1. www.opengroup.org/architecture/togaf8-doc/arch/toc.html (accessed April 4, 2009).

Tyree 2005. Tyree, Jeff, and Art Akerman. 2005. "Architecture Decisions: Demystifying Architecture." *IEEE Software* 22:2.

UML 2.2 2009. UML 2.2 Superstructure Specification. Object Management Group, Inc. Document number 09-02-02, February 2009. www.omg.org/docs/formal/09-02-02.pdf (accessed April 4, 2009).

UML User Guide 1999. Booch, Grady, James Rumbaugh, and Ivar Jacobson. 1999. *The Unified Modeling Language User Guide*. Reading, MA: Addison-Wesley.

von Halle 2001. von Halle, Barbara. 2001. *Business Rules Applied*. New York:

John Wiley and Sons.

Warmer 1999. Warmer, Jos, and Anneke Kleppe. 1999. *The Object Constraint Language: Precise Modeling with UML*. Reading, MA: Addison-Wesley.

Wirfs-Brock 1990. Wirfs-Brock, Rebecca, et al. 1990. *Designing Object-Oriented Software*. Englewood Cliffs, NJ: Prentice Hall.

Zachman 1987. Zachman, John. 1987. "A Framework for Information Systems Architecture." *IBM Systems Journal* 26:3.

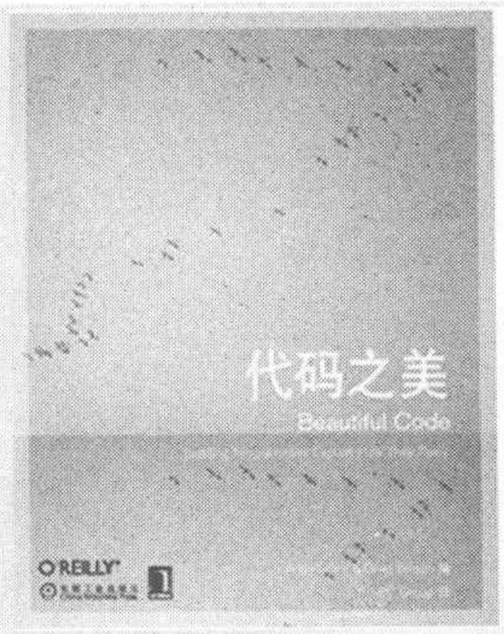
Zimmermann 2008. Zimmermann, Olaf, et al. 2008. "Combining Pattern Languages and Reusable Architectural Decision Models into a Comprehensive and Comprehensible Design Method." *Proceedings of 7th IEEE/IFIP Working Conference on Software Architecture*. Los Alamitos, CA: IEEE Computer Society.

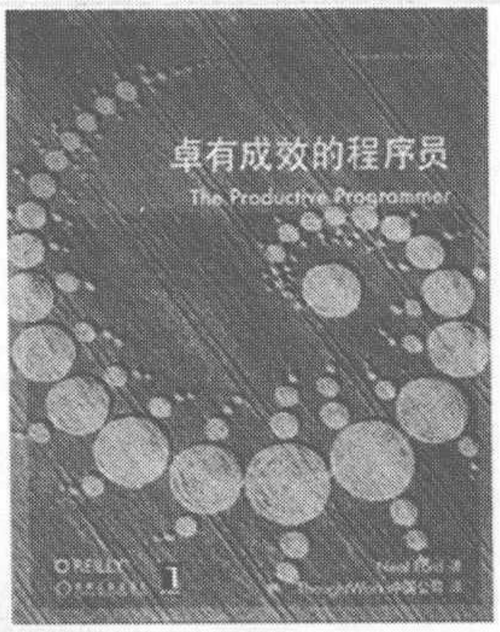
延伸阅读

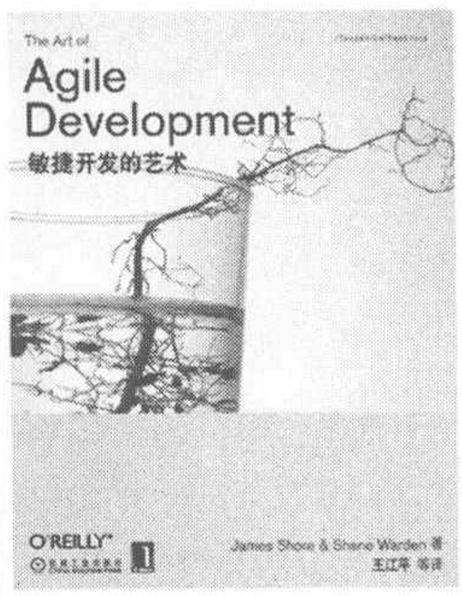
为每一个团队提供最有价值的阅读服务

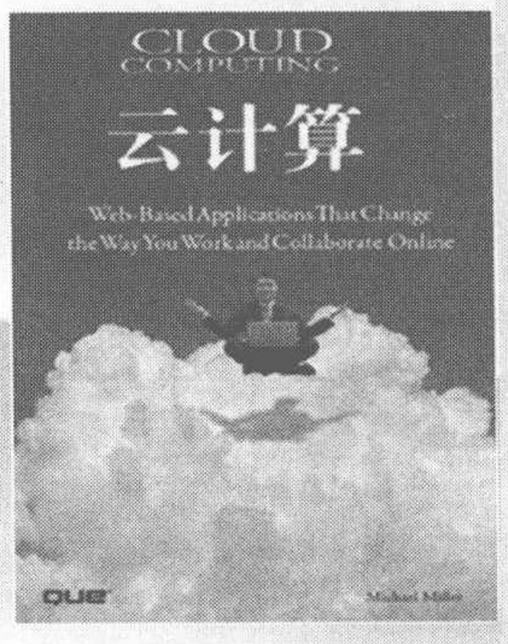
每一本书都值得您和您的团队一起阅读

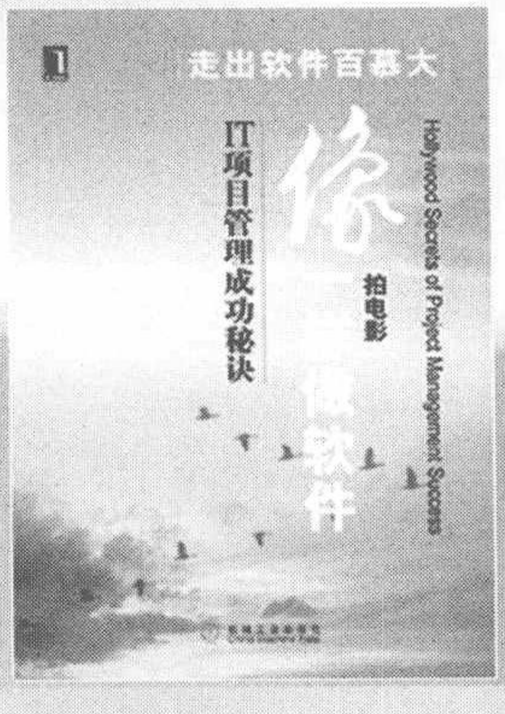
分享阅读分享成功

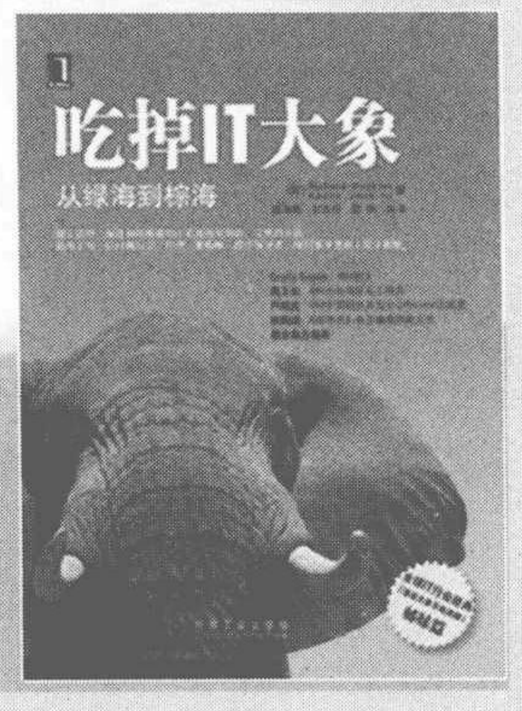












专业成就人生
立体服务大众

HZ BOOKS

www.hzbook.com

填写读者调查表　加入华章书友会
获赠精彩技术书　参与活动和抽奖

尊敬的读者：

感谢您选择华章图书。为了聆听您的意见，以便我们能够为您提供更优秀的图书产品，敬请您抽出宝贵的时间填写本表，并按底部的地址邮寄给我们（您也可通过www.hzbook.com填写本表）。您将加入我们的“华章书友会”，及时获得新书资讯，免费参加书友会活动。我们将定期选出若干名热心读者，免费赠送我们出版的图书。请一定填写书名书号并留全您的联系信息，以便我们联络您，谢谢！

书名：　　书号：7-111-(　　)

姓名：	性别：☐男　☐女	年龄：	职业：
通信地址：		E-mail：	
电话：	手机：	邮编：	

1. 您是如何获知本书的：

☐朋友推荐　☐书店　☐图书目录　☐杂志、报纸、网络等　☐其他

2. 您从哪里购买本书：

☐新华书店　☐计算机专业书店　☐网上书店　☐其他

3. 您对本书的评价是：

技术内容　☐很好　☐一般　☐较差　☐理由_____

文字质量　☐很好　☐一般　☐较差　☐理由_____

版式封面　☐很好　☐一般　☐较差　☐理由_____

印装质量　☐很好　☐一般　☐较差　☐理由_____

图书定价　☐太高　☐合适　☐较低　☐理由_____

4. 您希望我们的图书在哪些方面进行改进？

_____

_____

5. 您最希望我们出版哪方面的图书？如果有英文版请写出书名。

_____

_____

6. 您有没有写作或翻译技术图书的想法？

☐是，我的计划是_____　☐否

7. 您希望获取图书信息的形式：

☐邮件　☐信函　☐短信　☐其他_____

请寄：北京市西城区百万庄南街1号　机械工业出版社　华章公司　计算机图书策划部收

邮编：100037　电话：（010）88379512　传真：（010）68311602　E-mail: hzjsj@hzbook.com